



Bengt Jonsson

Compositional Verification of Distributed Systems

Ph. D. Thesis
Department of Computer Systems
Uppsala University
Uppsala, Sweden

ISSN 0283-0574



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41551502_0001

Bengt Jonsson

Compositional Verification of Distributed Systems



Bengt Jonsson

Compositional Verification of Distributed Systems

Ph. D. Thesis
Department of Computer Systems
Uppsala University
Uppsala, Sweden

ISSN 0283-0574

Abstract

Jonsson, B., 1987, Compositional Verification of Distributed Systems, *DoCS 87/09*, Uppsala, 142 pp. ISSN 0283-0574

We present a method for specification and verification of distributed systems. The method applies to systems that can always accept input messages from their environment. The method is compositional, and handles both safety properties and liveness properties,

A system is specified by its quiescent traces. A quiescent trace is a sequence of message transmissions and receptions in a computation after which the system will not transmit more messages unless extra input messages are supplied.

Properties of quiescent traces are specified by transition-systems with liveness requirements. A transition system generates traces in sequences of transitions that satisfy the liveness requirements. We present techniques for verifying that a system conforms to a specification, given that its components satisfy their specifications. Safety properties are verified by establishing simulations between transition systems. Liveness requirements are verified by induction over well-founded sets, using techniques similar to proof techniques for proving termination of programs.

The semantical foundations of the method are provided by a model which describes a system by the set of its quiescent traces. For nondeterministic Kahn-networks, we prove that our model is the minimal extension, in the sense of providing least information, of the model that describes a system by a relation between channel histories.

The method is illustrated by verifying three examples: A nondeterministic dataflow network, due to Brock and Ackermann, a sliding window protocol similar to the one in HDLC, and Thomas' algorithm for ensuring consistency in a distributed database.

*Bengt Jonsson, Department of Computer Systems, Uppsala University, Box 520,
S-751 20 Uppsala, Sweden, E-mail: bengt@kuling.UUCP*

© Bengt Jonsson 1987

Original produced with Apple laser printer

Printed in Sweden by

Direkt Offset, Nyström & Co AB, Uppsala 1987

ISSN 0283-0574

till Gun

Acknowledgments

I had the privilege to carry out part of my graduate studies at Stanford University, where Zohar Manna functioned as my advisor. For his generosity and competent advice, regarding both research and other aspects of life, I am deeply grateful. I wish to thank the people at the computer science department at Stanford for a wonderful working atmosphere. My stay was made possible by support from the Fulbright Commission, the Sweden-America Foundation, the grant fund of Thun and the foundation of Blanceflor Boncompagni-Ludovisi.

The original inspiration for this thesis was provided by work of Jay Misra and Mani Chandy. I wish to thank Jay Misra for explaining and discussing his ideas, and for encouraging me in pursuing this work. I am very indebted to Ernst-Rüdiger Olderog, who has generously shared his knowledge through suggestions and criticism, being of invaluable help during this work. Many valuable comments have been received during discussions with Smuel Katz and Amir Pnueli.

I am very grateful to my advisor, Björn Pehrson, who has supported and encouraged me through my graduate education. My colleague and friend Joachim Parrow has always been an inspiring discussion-partner. I thank him for all the work he has devoted to suggestions and criticism of this work.

Thanks to all people who have discussed and commented this work at various stages: Martin Abadi, Parosh Abdullah, Peter Dybjer, Per Gunningberg, Rune Gustavsson, Fredrik Orava, Viggo Stoltenberg-Hansen, Steffen Weckner. A special thanks to Parosh Abdullah for making the illustrations in this thesis. Thanks to all my colleagues at Uppsala University and at the Swedish Institute of Computer Science. All have assisted in some way.

Throughout my graduate education, I have been supported by the Swedish Board for Technical Development (STU) under contracts no. 82-3406, and 86-4250. Support has also been provided by NSF under contract DCR-85-12356, by DARPA under Contract NAAA39-84-C-0211, and by the USAF under Contract AFOSR-85-0383

Table of Contents

1	Introduction	11
2	Related Models	17
2.1	Introduction	17
2.2	Models of Distributed Systems	17
2.2.1	History models	18
2.2.2	Trace models	19
2.2.3	Extensions of trace models	20
2.2.4	Tree models	21
2.2.5	Petri nets and “true concurrency”	22
3	Modeling I/O-Systems	25
3.1	Introduction	25
3.2	I/O-systems	27
3.2.1	Definition of I/O-systems	27
3.2.2	Operations on I/O-systems	29
3.3	A Model of I/O-Systems	32
3.4	Comparison with Other Models	33
3.4.1	Preliminaries	34
3.4.2	Comparison between models of I/O-systems	35
3.5	Applications to Kahn-networks	39
3.5.1	Kahn-networks as I/O-systems	40
3.5.2	Comparison with history model	41
3.6	Proofs from Chapter 3	43

4	Specification and Verification	55
4.1	Introduction	55
4.2	Specification and Verification: a General Framework	56
4.2.1	Trace specifications	57
4.2.2	Proof rules	58
4.2.3	Completeness issues	61
4.3	Trace Generators	63
4.4	Reasoning About Trace Generators	67
4.4.1	Product of trace generators	67
4.4.2	Proving implication between trace generators	68
4.4.3	Proof rules for auxiliary formulas	72
4.5	Completeness Issues	76
4.6	Specifying Communication over FIFO Channels	79
4.6.1	Buffered trace generators	80
4.6.2	Reasoning about buffered trace generators	83
4.7	Proofs from Chapter 4	88
5	Applications	101
5.1	The Brock-Ackermann Counterexample	101
5.1.1	Informal description of the network	102
5.1.2	Specifications of components	103
5.1.3	Specification of the network	104
5.1.4	Verification of $S1$ and $S2$	105
5.1.5	Verification of $T1$ and $T2$	106
5.2	The Sliding Window Protocol of HDLC	108
5.2.1	Description of the protocol	108
5.2.2	Specification of data transfer	110
5.2.3	Specification of the components	111
5.2.4	Verification of the protocol	113
5.2.5	Discussion	118
5.3	Algorithm for Multiple Copy Databases	120
5.3.1	Informal description of the algorithm	121
5.3.2	specification of the components	122
5.3.3	Specification of the database	125
5.3.4	Verification of the algorithm	128
5.3.5	Discussion	133
6	Conclusion	135
	References	139

1 Introduction

The importance of distributed computer systems has increased significantly in the last decade. They now occur in many application areas, such as computer networks, databases, office-systems, etc. Due to the often complex patterns of interaction between components of a distributed system, the problem of establishing correct behavior becomes more difficult than for a system with centralized control. It is often difficult to explore all possible patterns of interaction by techniques such as testing and simulation.

A rigorous method for establishing correctness of a system is to formally specify properties of a system's intended behavior, and verify that the system conforms to this specification. Properties of a system's behavior are often classified into liveness properties and safety properties. Safety properties are of the form "nothing bad will ever happen," e.g., that the outputs produced are the right ones, or that the system does not deadlock. Liveness properties are of the form "something good must eventually happen," e.g., that the right output will be produced, or that the system will terminate.

Rather than specifying properties of systems directly, a common technique is to first construct a *model*, which describes the relevant aspects of a system while abstracting away irrelevant details, and then specify properties of the system's description in the model. For example, an often used model for sequential programs describes a program as a function or relation between its initial state and its final state. A program is then specified by specifying this relation, e.g., by pre- and postconditions. This is done e.g., in the work of Floyd ([Fl 67]), Hoare ([Ho 69]) and Dijkstra ([Di 76]).

The verification of a system is facilitated if the method for specification and verification is *compositional*. This means that the verification of a system can be split up into verifications of its components. With a compositional specification method, it is possible to reason about a system using specifications of its components, without bothering about how the components are implemented. If

1 Introduction

the specification method is based on a model, then the model's description of a system should be obtainable from the descriptions of its components.

Distributed systems are often designed to maintain a continuous interaction with their environment, receiving input and producing output at several instants during their execution. If systems communicate by messages, it seems natural to specify the possible sequences of messages that may be received and transmitted by the system. In the work by Chen and Hoare ([CH 81]), by Misra and Chandy ([MC 81]), and by Back and Mannila ([BM 82]), a system is modeled by its *traces*, and specified by properties of its traces. A trace is a sequence of message transmissions and receptions that can occur up to some point of a computation.

Consider for example the following systems *A* and *B* in Fig. 1.1. The system *A* outputs 0's repeatedly, without ever terminating. The system *B* receives 0's and for each received 0 transmits a 1 after an unspecified delay.



Fig. 1.1: The systems *A* and *B*.

The traces of *A* are the finite sequences of 0's. The traces of *B* are finite sequences of 0's and 1's, with at least as many 0's as 1's.

A nice thing about trace-specifications is that they are compositional. If the transmission of an output and its reception by other systems is one atomic operation, then the set of traces of a system can be obtained by "merging" traces of its components, while synchronizing on common messages. For instance, from the trace $\langle 0\ 0 \rangle$ of *A*, and the trace $\langle 0\ 1\ 0 \rangle$ of *B*, we obtain the trace $\langle 0\ 1\ 0 \rangle$ of the system consisting of both *A* and *B*.

A system's traces describe safety properties of a system, but not its liveness properties. For example, the traces of *A* represent the property "only 0's may appear as output." However, the liveness property "after each 0, a next 0 will appear," is not described by the traces. A reason is that another system A_{bad} which also outputs 0's but sometimes terminates after the 3rd one (e.g., if the weather is rainy) has the same set of traces but different liveness properties.

To overcome this limitation, Misra and Chandy ([Mis 84]) have proposed to specify a system by properties of its *quiescent traces*. A quiescent trace of a system is a sequence of inputs and outputs that in some computation will not be extended further by output of the system unless more input is supplied. In the

previous example, the quiescent traces of B are sequences that contain equally many 1's as 0's. Each infinite sequence of inputs and outputs is considered quiescent, since an infinite sequence cannot be extended. Therefore A has the quiescent trace 0^ω , consisting of an infinite sequence of 0's.

Note that a nondeterministic system may sometimes extend a quiescent trace by output. For instance, the system A_{bad} has the quiescent traces $\langle 0\ 0\ 0 \rangle$ and 0^ω , although it will sometimes extend the trace $\langle 0\ 0\ 0 \rangle$ (if the weather is not rainy). A quiescent trace only states that in some computations the trace will not be extended.

The set of quiescent traces describe safety properties, since the set of traces are obtained as prefixes of the quiescent traces, and liveness properties, since they describe when the traces must be extended. For instance, the different sets of quiescent traces of A and A_{bad} describe the different liveness properties of the two systems.

If the transmission of a message, and its reception by the receiving systems is one atomic operation, then the quiescent traces of a system can be obtained from those of its components in the same way as the traces of a system are obtained from the traces of its components. A prerequisite is that a system can transmit output to other systems whenever its trace is not quiescent. This prerequisite is satisfied by systems that communicate via an intermediate network, but for such systems the transmission and reception of a message is not one atomic operation. This limitation can be overcome by redefining what it means to "receive" a message: the reception of a message (an input) is considered to occur when the message is transmitted to the network by the sending system. After the atomic operation of transmitting and receiving a message, the message may still be in transit over the intermediate network.

Using the convention that input occurs when a message is transmitted to a system, Misra and Chandy therefore applied their ideas to systems that satisfy the following requirements:

- (1) A system can always accept input from other systems.
- (2) The transmission of a message by a system, and its reception by the receiving system(s) is an atomic operation that occurs simultaneously in the transmitting and receiving systems.

Examples of systems that satisfy these requirements are systems that communicate over an intermediate network to which messages can always be submitted, dataflow networks, systems that communicate by interrupts, and systems of hardware modules. Note that in frameworks such as CCS ([Mi 80]) and CSP ([Ho 78]), communication is only assumed to satisfy requirement (2).

1 Introduction

The presentation by Chandy and Misra ([Mis 84]) is informal, and does not address the question of fairness properties in connection with infinite quiescent traces. In our previous work ([Jo 85]), we formalized these ideas and their connection with fairness properties. This formalization was used as a basis for model and a verification method for asynchronously communicating systems.

In this thesis, we use the term *I/O-system* to refer to systems that satisfy requirements (1) and (2). The name comes from the work of Stark ([St 84]), who uses a similar definition of I/O-systems to define a notion of consistency for specifications.

In this thesis, we present

- A formal definition of I/O-systems, and a compositional model of I/O-systems, which describes a system by its quiescent traces.
- A comparison of the descriptive power of the model relative to other models of I/O-systems.
- A method for specification and verification of I/O-systems.
- Verification of both safety and liveness properties of two nontrivial distributed systems.

Our comparison of models relate models according to how much information about systems they abstract away. A model M_1 is considered more abstract than the model M_2 if a system's description in M_2 contains more information than its description in M_1 . It is then possible to map the description in M_2 to the description in M_1 , and we use the existence of such mappings to compare different models.

In our specification and verification method, we specify properties of quiescent traces by transition systems, with liveness requirements. The transition system generates traces by sequences of transitions that satisfy the liveness requirements.

To illustrate our specification formalism, we sketch a simple specification of the quiescent traces of the system B in Fig. 1.2. The specification is a transition system that records the number x of 0's for which no 1 has been transmitted. This number x is increased with each 0 received, and decreased with each 1 transmitted. Note that the 1 may be transmitted only if $x \geq 0$, and that an extra 1 must eventually be transmitted under the same condition.

Events:	0 , 1
Variables:	x
Initialization:	$x = 0$
Transitions:	The event 0 may always occur, causing x to be incremented by 1. The event 1 may occur if $x \geq 1$, causing x to be decremented by 1.
Liveness:	If $x \geq 1$, then eventually the event 1 must occur.

Fig. 1.2: Specification of the quiescent traces of B .

We present a verification method for proving a system's specification from specifications of its components. Safety properties are established by a simulation between the corresponding transition systems. Liveness properties are established by induction over well-founded sets.

Similar methods, in which specifications have the form of transition systems have been presented by Lamport ([La 83]) and Stark ([St 86]). Related work by Lynch and Tuttle is also in progress. Lamport uses transition systems to constrain the possible transitions of a system, whereas Stark uses them to specify sequences of communication events performed by a system. Both Lamport and Stark formulate liveness requirements in linear temporal logic. Their verification techniques involve proofs of formulas in temporal logic.

While temporal logic is well suited for specifying liveness properties, proof methods in temporal logic are not so well-developed as in classical logic. Methods for proving liveness properties that stay within classical logic by using induction over well-founded sets have been used by Floyd ([Fl 67]) to prove termination. This method has been extended to proving termination and other liveness properties under fairness conditions by Grumberg et al ([GFMR 81]), by Lehmann, Pnueli, and Stavi ([LPS 81]), by Owicki and Lamport ([OL 82]), and by Manna and Pnueli ([MP 84]). We adapt these methods to our verification method by allowing liveness requirements for which proof methods that stay within classical logic can be formulated.

The usefulness of a verification method is best judged by applying it to nontrivial examples. We therefore apply our method to three well-known examples: a dataflow-like network due to Brock and Ackermann ([BA 81]) the sliding window protocol of HDLC, and a voting protocol for concurrency control in replicated databases, due to Thomas ([Th 79]). The two latter examples are nontrivial, and indicate that our verification method can be used for rather complex problems.

Organization of the Thesis

In chapter 2, we review related work on modeling of distributed systems. The literature in this area is large, and impossible to survey within the scope of this thesis. We therefore restrict the chapter to models that in some form or another use sequences of message transmissions to model distributed systems. We point out connections between the class of systems that are modeled and the structure of the model that is used.

In chapter 3, we develop the semantical foundations of the method by presenting a model of I/O-systems. In section 3.2, we formally define I/O-systems. The definition uses a transition system formalism. The definition also clarifies our approach to the problem of fairness. We define operations for composing I/O-systems, and for abstracting information about the behavior of an I/O-system. In section 3.3, we define the model which describes a system by its quiescent traces. We prove that the model is compositional, by establishing rules for the operations defined in section 3.2.

In sections 3.4 and 3.5, we investigate the existence of other compositional models of I/O-systems, and their relation to the model defined in section 3.3. We relate models by the amount of information they provide about an I/O-system. In section 3.5, we restrict the attention to nondeterministic Kahn-networks, i.e., systems that communicate via unbounded FIFO channels. These systems were studied by Kahn ([Ka 74]), who presented an elegant model for *deterministic* Kahn-networks. The main result is that for nondeterministic Kahn-networks our model is minimal, in the sense that it contains less information than any other compositional model that extends Kahn's model to nondeterministic networks.

In chapter 4, we consider specification and verification of I/O-systems. In section 4.2, we consider the general question of specifying a system by its quiescent traces, without treating the problem of how to express properties of quiescent traces. In section 4.3, we present a formalism for expressing properties of sequences by *trace generators*, i.e., transition systems with liveness requirements. In section 4.4, we present methods for verifying systems that are specified by trace generators. In section 4.5, we discuss completeness of the rules. In section 4.6, we present techniques for specification and verification of systems that communicate via FIFO channels.

In chapter 5, the verification method is illustrated by applications to three well-known examples: the counterexample of Brock and Ackermann [BA 81], the sliding window protocol found in HDLC, and Thomas' algorithm for concurrency control in replicated databases ([Th 79]). Chapter 6 contains conclusions, comparisons with related approaches to specification and verification of distributed systems, and directions for future research.

2 Related Models

2.1 Introduction

The literature on modeling, specification, and verification of distributed systems is large and expanding. It is impossible to survey all approaches to modeling, specification, and verification of distributed systems within the scope of this thesis. An overview of approaches to the modeling of distributed systems has been written by Løvengreen ([Lø 85]).

Here we survey models of distributed systems that are related to our model. Our model describes a system by sequences of message transmissions. We therefore restrict the treatment to models that in some form use sequences of message transmissions to describe systems. In order to concentrate on certain aspects of a model, we present simplified views of some of the approaches.

2.2 Models of Distributed Systems

A sequential program which receives input at the beginning of its execution and produces output after completing its execution can be modeled by a function from its initial state to its final state if it is deterministic, or by a relation between input state and output state if it is nondeterministic. This view is the basis e.g., for the work of Floyd ([Fl 67]), Hoare ([Ho 69]), Dijkstra ([Di 76])). This view is also employed in many works on verification of concurrent programs, e.g., by Owicki and Gries ([OG 76]), and by Apt, Francez, and de Roever ([AFR 80]).

Distributed systems are often intended to maintain a continuous interaction with their environment, receiving several inputs and producing several outputs during their execution. Such systems can be modeled by sequences of interactions with the environment. In this section, we present models that are based on this view. For different classes of distributed system, different models have been proposed.

2.2.1 History models

One of the earlier approaches to modeling of distributed computing systems is due to Kahn ([Ka 74]). He considered systems that communicate by passing messages over directed FIFO channels. The channels deliver messages in the same order as they are sent, after some unspecified finite delay. We shall use the term *Kahn-networks* for this class of systems. Fig. 2.2.1 shows a Kahn-network with three components connected with each other and with the environment by channels.

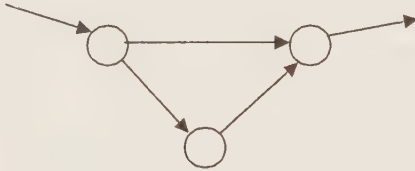


Fig. 2.2.1. A Kahn-network.

A *history* of a channel is the sequence of messages that is transmitted over the channel during a completed execution of the network. If each component follows a deterministic program, then the histories of the input channels uniquely determine the histories of the output channels. Kahn proposed to model a deterministic Kahn-network by a function from histories of its input channels to the histories of its output channels. For example, a network which copies data from its only input channel to its only output channel is described by the identity function on histories.

Kahn showed that the function that models a network can be obtained by composition of the functions that model its components. If the structure of the network contains cycles, the composition also involves a fixedpoint construction. Thus the model by Kahn is compositional for the class of deterministic Kahn-networks.

Nondeterministic networks cannot be modeled by functions. The obvious generalization is to model a nondeterministic network by a *relation* between the histories of its channels. However, Brock and Ackermann ([BA 81]) have shown that this model fails to be compositional. They proved this by constructing a nondeterministic network and replacing one of its components by another component modeled by the same relation between its histories. They obtained a network with a different history relation than the original network. Thus the example shows that history relations of components do not uniquely determine the history relation of a network. The counterexample of Brock and Ackermann is presented in section 4.1 of this thesis.

Brock and Ackerman also concluded that the reason for the inadequacy of history relations is that they do not contain information about the temporal ordering between occurrences of message transmissions and receptions on different channels. They proposed an extension of the history model, called “scenarios,” in which each tuple in the history relation is provided with an ordering relation between the individual elements of the histories of the different channels.

Pratt ([Pr 82]) observed that a scenario can be regarded as a partially ordered multiset (a multiset is a set in which each element can have several occurrences). Thus, he proposed to model a network by a set of partially ordered multisets. Pratt has subsequently developed these ideas ([Pr 84], [Pr 85]). A model that employs partially ordered multisets has also been proposed by Staples and Nguyen ([SN 85]). Other models for nondeterministic Kahn-networks have been proposed by Boussinot ([Bo 82]), Broy ([Broy 83]), Keller ([Ke 78]), Keller and Panangaden ([KP 84]), Kok ([Ko 86]), Kosinski ([Ko 78]), Park ([Pa 83]).

2.2.2 Trace models

The missing ordering information in the history model can be supplied by ordering all message transmissions and receptions in a computation into a linear sequence. An often used model for distributed systems is based on traces. A *trace* is a sequence of message transmissions and receptions that may occur up to some point of a computation.

Models that represent a system by its traces have been proposed by Back and Mannila ([BM 82]), by Chen and Hoare ([CH 81]), and by Misra and Chandy ([MC 81]). The transmission of a message, and its reception by the receiving systems is considered as an atomic operation, which occurs simultaneously in the transmitting and receiving systems. The traces of a system can then be obtained by merging traces of its components, synchronizing on common messages.

If the transmission of a message takes a non-negligible amount of time, the transmission and reception of a message is not an atomic operation. However, it is possible to redefine what it means to “receive” a message, and consider the reception operation to occur when the message is transmitted to the system. By this convention, transmission and reception refer to the same atomic operation, and traces can be composed by synchronized mergings.

Misra and Chandy ([Mis 84]) observed that by this convention, traces can also be used to model liveness properties of systems that can always receive messages, using the idea of quiescence, described in chapter 1. In our previous work ([Jo 85]), we formalized this idea and used it for specification and verification.

In an earlier proposal for modeling liveness properties, Misra, Chandy and Smith ([MCS 82]) extended a trace model by designating certain traces as “non-quiescent.” A non-quiescent trace is a trace that must be extended by output of

the system. This extension permits modeling of certain liveness properties. The model does not include infinite traces, and cannot model properties pertaining to fairness.

A model for systems that communicate via shared variables, which is related to ours, has been presented by Lynch and Fischer ([LF 81]). A system is modeled by the sequences of operations on variables that can occur in completed computations. They obtain a similar rule for obtaining the description of a system from descriptions of its components.

2.2.3 Extensions of trace-models for synchronous communication

For systems that communicate by rendez-vous, and where a system can not always receive input, traces do not contain enough information to capture liveness properties. To illustrate this point, consider three systems S_1 , S_2 , and S_3 , that are defined by the state-transition diagrams in Fig. 2.2.2.

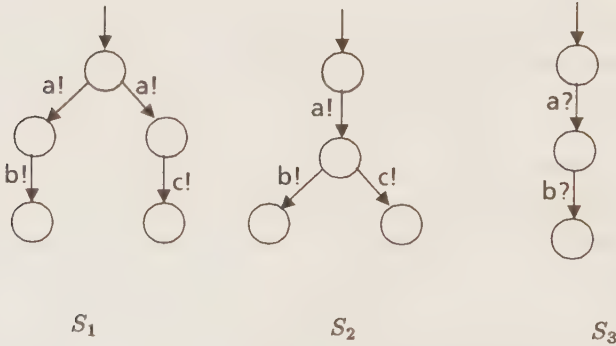


Fig. 2.2.2. State-transition diagram description of S_1 , S_2 , and S_3 .

The systems communicate by synchronized message-passing over the channels a , b , and c . The label $a?$ denotes the reception of a message on channel a , and $a!$ denotes the transmission of a message over channel a .

The systems S_1 and S_2 cannot be distinguished by their possible sequences of communications. Both can perform the sequences of communications $\langle a!, b! \rangle$ and $\langle a!, c! \rangle$, or prefixes of them. However, when either S_1 or S_2 is connected with the system S_3 , and we require that all communications via a or b be done between two components, then the liveness properties of the resulting systems are different, depending on whether we connect S_1 or S_2 with S_3 . The composition of S_1 and S_3 can always perform the sequence $\langle a, b \rangle$. The composition of S_1 and S_3 can also perform such a trace, but it can also deadlock after having communicated only via a , if S_2 traverses the right branch in its state-transition diagram when transmitting a message over a .

The above example shows that liveness properties of a system cannot be deduced from only the possible sequences of communication events of its components. We also need information about the situations in which a system may deadlock when composed with other systems. In the above example, a description of S_2 must state that the system may deadlock after having communicated via a , if its environment cannot communicate via b , and that in another case it may deadlock after having communicated via a , if its environment cannot communicate via c .

To obtain compositionality for synchronous systems, and to model potential deadlocks of a system, extensions of trace models have been proposed in which information about potential deadlocks is added to each trace. One such model is the *readiness model* by Hoare and Olderog ([Ho 81], [OH 84]). This model also contains information about the set of communication events that are possible after performing a sequence of communications. In the above example, the difference between S_1 and S_2 is captured by the fact that S_1 can perform the sequence $\langle a! \rangle$, and thereafter communicate via both b and c , whereas S_2 can either perform $\langle a! \rangle$ and communicate via b , or perform $\langle a! \rangle$ and communicate via c .

Other models that in some form or another include information about possible communications after a trace are the failure model by Brookes, Hoare and Roscoe ([BHR 84], [Brookes 83]), the acceptance trees by Hennessy ([He 85]), and the model by Francez, Lehmann and Pnueli ([FLP 84]).

The readiness model and the failure model can describe certain types of liveness properties, but not properties relating to fairness, since they only consider finite sequences of communication events. Nguyen et al ([NDGO 86]) have presented a model, which also captures fairness properties. They describe a system by a set of behaviors. A behavior can be regarded a sequence of communication events that occur in a completed, possibly infinite, computation. At each point of this sequence, the behavior also contains information about possible communication events.

2.2.4 Tree models

The models presented in the preceding subsection consist of information about individual computations of a system. For instance, the set of traces describes the sequences events that may occur in individual computations. The models do not relate these individual computations to each other. For instance, if a system can perform the sequences $\langle a!, b! \rangle$ and $\langle a!, c! \rangle$, then it can either perform $a!$, and then decide whether to perform $b!$ or $c!$, or it can already have made this decision before performing the first $a!$. Consider the system S'_2 which is defined in the following state-transition diagram.

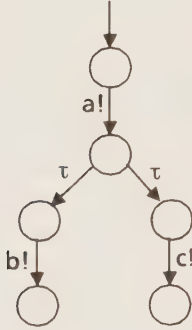


Fig. 2.2.3. The system S'_2 .

Transitions labeled by τ denote some internal computation of a system which leads from one state to another. The system S'_2 is not distinguished from S_2 in the readiness-model, since the system S'_2 can deadlock when composed with other systems in the same way as S_2 can. However, the choice between performing $b!$ or $c!$ is now made after performing $a!$.

In Milner's CCS ([Mil 80]), a system is modeled as a tree. The branching structure of the tree reflects the possible nondeterministic choices that can be made at each point of a computation. Examples of tree models are the state-transition diagrams of S_1 and S_2 in Fig. 2.2.2. Milner describes how a system can be observed from the exterior, and uses this to define an equivalence between synchronization trees. Two trees are considered equivalent if they cannot be distinguished by external observation. It should be noted that S_2 and S'_2 are distinguished by this equivalence, the reason being that the nondeterministic choice is either before or after the communication via a .

2.2.5 Petri nets and "true concurrency"

So far, the models described in the previous sections (except the models that employ partial orders) do not contain any information about the degree of concurrency of a system. They do not state whether a sequence of events is produced by one component or by several components executing more independently. A system with two independent components, where one component performs the sequence $\langle a! \rangle$ and the other component performs the sequence $\langle b! \rangle$, is regarded as equivalent to a system with only one component which performs either of the sequences $\langle a!, b! \rangle$ or $\langle b!, a! \rangle$.

In some approaches, this identification is found unsatisfactory. For instance,

Pratt ([Pr 82]) argues that if a and b denote transmissions that extend over time and occur concurrently, then we cannot model this by saying that they occur in either order.

One of the earliest approaches to modeling concurrent activities is the area of Petri Nets ([Br 79], [Pe 81]). Other approaches are the event structures of Winskel ([Wi 80]), Pratt's use of partially ordered multisets ([Pr 82], [Pr 84]), the work by Mazurkewicz ([Ma 84]), and by Boudol and Castellani ([BC 87]).

3 Modeling I/O-Systems

3.1 Introduction

In this chapter, we shall present a model for a class of distributed systems, called I/O-systems. Intuitively, I/O-systems communicate by message-passing, and satisfy the following two requirements:

- (1) A system can always accept input from other systems.
- (2) The transmission of a message by a system, and its reception by the receiving system(s) is an atomic operation that occurs simultaneously in the transmitting and receiving systems.

When modeling distributed systems in our framework, we satisfy these requirements by defining what it means to “receive” a message as follows: the reception of a message (an input) is considered to occur when the message is transmitted to the network by the sending system. After the atomic operation of transmitting and receiving a message, the message may still be in transit over the intermediate network.

Our model describes an I/O-system by its quiescent traces, i.e., sequences of inputs and outputs that in some computation will not be extended further by output of the system unless more input is supplied.

We use a formal definition of I/O-systems to define the quiescent traces of our model. This definition regards I/O-systems as transition systems. A transition system has a state, which can be changed by transitions. Message transmissions and receptions can also occur simultaneously with a transition. Transition systems are often used for operational descriptions of computing systems, e.g., by Plotkin ([Pl 81]), and Manna and Pnueli ([MP 81]).

Our model of I/O-systems considers infinite computations, and we shall therefore consider the question of fairness. For example, consider a system consisting of two components *A* and *B* that execute independently in parallel.

Assume that the relative speeds of A and B is either unknown, or irrelevant for the purpose of describing the system. The only relevant property of the execution speeds is the fairness property that both A and B will, if possible, perform infinitely many execution steps if the system performs infinitely many execution steps.

We represent fairness properties of I/O-systems by *fairness sets*. Fairness sets have been used to represent fairness properties of transition systems, e.g., by Manna and Pnueli ([MP 81]). A fairness set is a set of transitions which must not be neglected indefinitely in an infinite execution of the system. We shall use fairness sets to represent activities of the system that must not be neglected in executions of the system. Since an I/O-system does not control the occurrence of input, transitions that cause input are not considered as part of any activity of the system.

A similar definition of I/O-systems has been given by Stark ([St 84]), who uses I/O-systems to define a notion of consistency for specifications. In his work on consistency, Stark also establishes results that are related to some of ours.

Having defined I/O-systems, we define operations for composing I/O-systems, and for abstracting away certain message-transmissions. By establishing rules for the operations in our model, we prove that the model is compositional with respect to these operations.

We also compare our model to some other models of I/O-systems with respect to descriptive power. Depending on what aspects of a system are considered interesting, different models may be appropriate for describing them. Models can be related to each other according to how much information they provide about I/O-systems. For instance, the quiescent traces of an I/O-system contain more information than the finite traces, since the finite traces can be obtained as prefixes of the quiescent traces. Back and Mannila ([BM 85]) discuss these questions in general and their connection to compositional proof systems. They apply the results to a trace model. Related work in a different framework has been done by Olderog and Hoare ([OH 84]), who present several models for communicating sequential processes, and related them to each other.

In this chapter, we present some other models of I/O-systems, and relate them to our model. We also relate our model to the history model for nondeterministic Kahn-networks. Kahn ([Ka 74]) has proposed to model deterministic Kahn-networks by histories of their channels. However, the straight-forward generalization to nondeterministic Kahn-networks, which describes a network by a relation between histories, is not compositional. The main result is that our model is the minimal compositional extension (in the sense of containing least information about a system) of the history model for nondeterministic Kahn-networks.

In the next section, we give formal definitions of the class of I/O-systems, and of the composition and abstraction operations on I/O-systems. In section 3.3, the quiescent traces of an I/O-system is defined as a model of I/O-systems. We prove that the model is compositional. Section 3.4 contains a presentation of some other compositional models of I/O-systems and their relationship to each other and to the model presented in section 3.3. In section 3.5, we consider the class of Kahn-networks, and relate the model presented in section 3.3 to the history model presented by Kahn ([Ka 74]). Proofs of some of the theorems in chapter 3 are collected in section 3.6.

3.2 I/O-Systems

3.2.1 Definition of I/O-systems

In our formalization of I/O-systems, we represent the transmission of a message by a *communication event*. We assume that communication events can be regarded as atomic operations. The communication events of an I/O-system are divided into *input events* that represent transmissions of messages from the environment to the system, and *output events* that represent transmissions of messages by the system itself, both between components of the system and to its environment.

Definition 3.2.1. An *I/O-system* is a tuple $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$, where

I is a set of *input events*, not containing the silent event τ .

O is a set of *output events*, disjoint from I , not containing τ .

Σ is a set of *states*.

σ^0 is the *initial state* of the system.

R is a subset of $\Sigma \times (I \cup O \cup \{\tau\}) \times \Sigma$, called the set of *labeled transitions*.

A labeled transition in R is denoted by $\sigma_1 \xrightarrow{e} \sigma_2$, where $\sigma_1, \sigma_2 \in \Sigma$ and $e \in (I \cup O \cup \{\tau\})$.

$\mathcal{F}$ is a finite collection of *fairness sets*. Each fairness set is subset of R .

An I/O-system $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$ satisfies the following properties:

- 1) For each state $\sigma \in \Sigma$ and input event $e \in I$ there is a state $\sigma' \in \Sigma$ such that $\sigma \xrightarrow{e} \sigma' \in R$.
- 2) No fairness set $F \in \mathcal{F}$ contains any transition $\sigma \xrightarrow{e} \sigma'$ for which $e \in I$
- 3) Each transition $\sigma \xrightarrow{e} \sigma'$ for which $e \in (O \cup \{\tau\})$ is a member of some fairness set in $\mathcal{F}$.

□

Definition 3.2.2. A transition $\sigma \xrightarrow{e} \sigma'$ is called an *input transition*, *output transition*, or *silent transition*, depending on whether e is an input event, an output event, or the silent event τ .

A transition $\sigma \xrightarrow{e} \sigma'$ is *enabled* in the state σ . A fairness set F is enabled in σ if a transition in F is enabled in σ .

□

Intuitively, the state of an I/O-system contains the information that is necessary for characterizing its possible future behavior. Typically, the state contains the contents of internal variables, message buffers, etc. The set of transitions define how the state may change. Some transitions are simultaneous with communication events, and are labeled by the corresponding communication event. Other transitions merely update the internal state, and do not occur with any communication event. These transitions are labeled with the silent event τ . Each fairness set is intended to denote the transitions that describe the actions of a component which must not be neglected indefinitely in an infinite execution of the system.

Requirement 1) formalizes the assumption that an I/O-system can always accept input. Requirements 2) and 3) reflect the view that a fairness set represents transitions of some activity that must not be neglected in an infinite computation. Requirement 2) then reflects the view that input is not caused by any activity of the system, since the system does not control the occurrence of input. Requirement 3) formalizes the assumption that the actions of the system itself is represented by the silent and output transitions of the system.

Definition 3.2.3. Let N be the I/O-system $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$. A *computation* of N is a finite or infinite sequence of transitions

$$\sigma^0 \xrightarrow{e^1} \sigma^1 \xrightarrow{e^2} \dots \xrightarrow{e^n} \sigma^n \xrightarrow{e^{n+1}} \dots$$

which fulfills the following conditions:

- (A) *Initialization.* σ^0 is the initial state of N .
- (B) *State to state sequencing.* Each transition $\sigma^{i-1} \xrightarrow{e^i} \sigma^i$ is a member of R .
- (C) *Fairness.* If the sequence is infinite, it must contain infinitely many occurrences of transitions from each fairness set $F \in \mathcal{F}$ which is enabled infinitely often.
- (D) *Quiescence.* If the sequence is finite, then no output transitions or silent transitions are enabled in its last state.

A *partial computation* is a finite or infinite sequence of transitions which fulfills conditions (A) and (B). □

Example 3.2.4. In this example, we define a particular I/O-system: a simple buffer, which receives messages from a channel α and transmits then onto channel β in the same order. We assume that the messages belong to a set M . The transmission of a message over α or β is a communication event denoted by $\alpha(m)$ or $\beta(m)$, where $m \in M$. The behavior can be modeled as the I/O-system $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$, where

$$I = \{\alpha(m) \mid m \in M\}.$$

$$O = \{\beta(m) \mid m \in M\}.$$

$\Sigma = M^*$, the set of finite sequences of elements in M . Intuitively, the state is the sequence of messages that have been received on α , but not transmitted on β .

$\sigma^0 = \langle \rangle$, the empty sequence.

R contains, for each state $\sigma \in \Sigma$ and message $m \in M$, the following two types of transitions (we use $\bullet$ to denote concatenation):

$$(i) \quad \sigma \xrightarrow{\alpha(m)} \sigma \bullet m$$

$$(ii) \quad m \bullet \sigma \xrightarrow{\beta(m)} \sigma$$

Events on α cause messages to be appended to the state, and events on β cause messages to be deleted from the state of the transition system.

$\mathcal{F} = \{F\}$, where F contains all transitions in R of form (ii). In other words, the process will eventually deliver all messages that have been received.

To verify that this is indeed an I/O-system, we check the three conditions:

- 1) A transition with an event in I , i.e., a transition of type (i). is always enabled.
- 2) The fairness set F does not contain any input transition of form (ii).
- 3) The fairness set F contains all output transitions, i.e., the transitions of form (ii).

□

3.2.2 Operations on I/O-systems

In this subsection, we define the following operations on I/O-systems:

- The *composition* of the I/O-systems $N_1, \dots, N_k$, denoted by $N_1 \parallel \dots \parallel N_k$, yields an I/O-system in which $N_1, \dots, N_k$ are components.
- The *abstraction* of a set E of output events of a system N , denoted by $N \setminus E$, yields an I/O-system which can no longer use the messages in E for communication with its environment.

Definition 3.2.5. The I/O-systems $N_i = \langle I_i, O_i, \Sigma_i, \sigma_i^0, R_i, \mathcal{F}_i \rangle$ for $i = 1, \dots, k$ are *compatible* if $O_i \cap O_j = \emptyset$ whenever $i \neq j$.

The *composition* of the compatible I/O-systems $N_i = \langle I_i, O_i, \Sigma_i, \sigma_i^0, R_i, \mathcal{F}_i \rangle$ for $i = 1, \dots, k$ is the tuple $N = \langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$, where

$O = \bigcup_{i=1}^k O_i$, i.e., the union of the set of output events of the components.

$I = \bigcup_{i=1}^k I_i - O$, i.e., the set of input events which are not output events.

$\Sigma = \Sigma_1 \times \dots \times \Sigma_k$, i.e., the cartesian product of the sets of states of the components. A state $\sigma \in \Sigma$ is a k -tuple, denoted as $\sigma = \langle \sigma_1, \dots, \sigma_k \rangle$.

$\sigma^0 = \langle \sigma_1^0, \dots, \sigma_k^0 \rangle$

R is obtained from the transitions in $R_1, \dots, R_k$ by the following rules.

(i) For silent transitions the rule is

$$\frac{\sigma_i \xrightarrow{\tau} \sigma'_i}{\langle \sigma_1, \dots, \sigma_i, \dots, \sigma_k \rangle \xrightarrow{\tau} \langle \sigma_1, \dots, \sigma'_i, \dots, \sigma_k \rangle}$$

i.e., silent transitions only concern one component σ_i of σ .

(ii) If $e \in I \cup O$ is a communication event, then each transition in R labeled by e involves a transition of each component which has e as a communication event. more precisely, for $e \in I \cup O$, let $N_{i_1}, \dots, N_{i_m}$ be the I/O-systems for which $e \in I_{i_j} \cup O_{i_j}$. Let $\sigma_{i_j} \xrightarrow{e} \sigma'_{i_j}$ be a transition of N_{i_j} labeled by e . Then there is a transition of N , which is obtained by the rule

$$\frac{\sigma_{i_1} \xrightarrow{e} \sigma'_{i_1} , \quad \dots , \quad \sigma_{i_m} \xrightarrow{e} \sigma'_{i_m}}{\langle \sigma_1, \dots, \sigma_{i_1}, \dots, \sigma_{i_m}, \dots, \sigma_k \rangle \xrightarrow{e} \langle \sigma_1, \dots, \sigma'_{i_1}, \dots, \sigma'_{i_m}, \dots, \sigma_k \rangle}$$

$\mathcal{F}$ is obtained from $\mathcal{F}_1, \dots, \mathcal{F}_k$ as follows: For each fairness set $F_i \in \mathcal{F}_i$, there is a fairness set $F \in \mathcal{F}$ consisting of the set of all transitions obtained from a transition in F_i by the rules (i) or (ii).

□

Intuitively, the state of N is a tuple, whose components are the states of the systems $N_1, \dots, N_k$. The transitions of N describe how this state changes. A transition of N is either

(i) an action of a component N_i with no simultaneous communication event. Such a transition does not affect other components, and is performed in isolation.

- (ii) an action with a communication event. Such a transition affects all components that can receive or transmit the corresponding message. This corresponds to a simultaneous action of the concerned components. Note that each communication is an output event of at most one component. This follows from the requirement that $N_1, \dots, N_k$ be compatible.

The fairness sets of N represent actions of subparts that must not be neglected indefinitely in infinite computations. These subparts of N are the union of the subparts of $N_1, \dots, N_k$ since N is just the composition of $N_1, \dots, N_k$. Each fairness set of some N_i is therefore mapped in a natural way onto a fairness set of N .

Note that the composition operator is “fair” with respect to the output transitions and silent transitions of each argument. This follows from the fact that all output and silent transitions of each N_i constitute a subset of $\mathcal{F}$.

Definition 3.2.6. Let $N = \langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$ be an I/O-system, and let $E \subseteq O$ be a set of output events of N . The *abstraction* of N with respect to E is the tuple $N \setminus E = \langle I, (O - E), \Sigma, \sigma^0, R \setminus E, \mathcal{F} \setminus E \rangle$ where

$R \setminus E$ is obtained from R by the following rules:

- (i) For each event $e \notin E$

$$\frac{\sigma \xrightarrow{e} \sigma'}{\sigma \xrightarrow{e} \sigma'}$$

- (ii) For each event $e \in E$

$$\frac{\sigma \xrightarrow{e} \sigma'}{\sigma \xrightarrow{\tau} \sigma'}$$

i.e., transitions labeled by events in E become silent, and other transitions are unchanged.

$\mathcal{F} \setminus E$ is a “copy” of $\mathcal{F}$, i.e., for each fairness set $F \in \mathcal{F}$ there is a fairness set $F \setminus E \in \mathcal{F} \setminus E$ containing all transitions derived from a transition in F according to the rules for obtaining $R \setminus E$.

□

Intuitively, the effect of the abstraction operation is that a set of communication events can no longer be used for communication with the environment. Formally, we achieve this by replacing abstracted events by the silent event τ . The abstraction operation is only defined with respect to output events. The motivation is that in practice, the abstraction operation is most useful for abstracting events that are used for communication between components of the system, and not for communication with the environment. Such events correspond to output events.

The following proposition establishes that the operations on I/O-systems defined above indeed result in an I/O-system.

Proposition 3.2.7.

- If the I/O-systems $N_1, \dots, N_k$ are compatible, then their composition is an I/O-system.
- If E is a set of output events of the I/O-system N , then the abstraction $N \setminus E$ is an I/O-system.

Proof. The proof is a straight-forward check of the conditions in definition 3.2.1. The details are carried out in section 3.6. $\square$

3.3 A Model of I/O-Systems.

In this section, we formally define the model of I/O-systems, describing a system by the set of its quiescent traces.

Definition 3.3.1. Let $N = \langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$ be an I/O-system.

- A *quiescent trace* of N is the sequence of communication events (i.e., non- τ events) in a computation of N .
- The denotation of N , denoted $\llbracket N \rrbracket$, is the triple $\langle I(N), O(N), Q(N) \rangle$, where
 $I(N) = I$, the set of input events of N
 $O(N) = O$, the set of output events of N
 $Q(N)$ is the set of quiescent traces of N .

$\square$

We shall prove that this model is compositional with respect to composition and abstraction. This means that $\llbracket N_1 \rrbracket \dots \llbracket N_k \rrbracket$ can be defined in terms of $\llbracket N_1 \rrbracket, \dots, \llbracket N_k \rrbracket$ and that $\llbracket N \setminus E \rrbracket$ can be defined in terms of $\llbracket N \rrbracket$.

Notation. Let E be a set of communication events.

$q[E]$ denotes the restriction of the sequence of communication events q to the set of communication events E , i.e., the subsequence of q containing only events in E .

$q \setminus E$ denotes the subsequence of the sequence of communication events q that are not in the set of communication events E . i.e., the result of deleting the events in E from q .

E^* denotes the set of finite sequences of events in E .

E^ω denotes the set of infinite sequences of events in E .

$E^\dagger$ denotes the set of finite and infinite sequences of events in E .

Theorem 3.3.2.

- (i) Let $N_1, \dots, N_k$ be compatible I/O-systems, and let N be their composition $N_1 \parallel \dots \parallel N_k$. The denotation $\llbracket N \rrbracket$ of N is the triple $\langle I(N), O(N), Q(N) \rangle$ where

$$O(N) = \bigcup_{i=1}^k O(N_i)$$

$$I(N) = \bigcup_{i=1}^k I(N_i) - O(N)$$

$$Q(N) = \{q \in (I(N) \cup O(N))^\dagger \mid q[(I(N_i) \cup O(N_i)) \in Q(N_i) \text{ for all } i]\}$$

- (ii) Let N be an I/O-system. Let E be a subset of $O(N)$. The denotation $\llbracket N \setminus E \rrbracket$ of $N \setminus E$ is the triple $\langle I(N \setminus E), O(N \setminus E), Q(N \setminus E) \rangle$ where

$$I(N \setminus E) = I(N)$$

$$O(N \setminus E) = O(N) - E$$

$$Q(N \setminus E) = \{q \setminus E \mid q \in Q(N)\}$$

Proof. The proof is given in section 2.8.

□

The theorem states that the quiescent traces of the composition N of $N_1 \parallel \dots \parallel N_k$ are precisely those which when restricted to the events of N_i are quiescent traces of N_i . Put another way, the quiescent traces of N are the possible “synchronized merges” of the quiescent traces of $N_1, \dots, N_k$. The quiescent traces of $N \setminus E$ are obtained by deleting events in E from the quiescent traces of N .

Example 3.3.3. As an illustration, consider the two I/O-systems N_1 and N_2 , where $\llbracket N_i \rrbracket = \langle \emptyset, \{e_i\}, Q_i \rangle$.

If $Q_1 = e_1^\omega$, and $Q_2 = e_2^\omega$, that is, both N_1 and N_2 perform an infinite sequence of events, then their composition is denoted by $\llbracket N_1 \parallel N_2 \rrbracket = \langle \emptyset, \{e_1, e_2\}, Q(N_1 \parallel N_2) \rangle$ where $Q(N_1 \parallel N_2)$ contains the fair merges of e_1^ω and e_2^ω . That is, each component performs infinitely many events.

On the other hand, if $Q_1 = e_1^\omega$, and $Q_2 = e_2^*$, that is, N_2 only performs an arbitrary finite number of events, then in the composition $\llbracket N_1 \parallel N_2 \rrbracket = \langle \emptyset, \{e_1, e_2\}, (e_1 \cup e_2)^* e_1^\omega \rangle$. That is, N_2 performs finitely many events whereas N_1 performs infinitely many events.

□

3.4 Comparison with Other Models

In section 3.3, we presented a compositional model of I/O-systems. This model is, of course, not the only model of I/O-systems. Depending on the

properties of a system that are considered interesting, different models may be appropriate. Due to the abundance of models of distributed systems, it is interesting to relate them to each other.

The purpose of a model is to describe the interesting properties of systems, while abstracting away uninteresting details. A “good” model describes the interesting properties and no extra details. We can therefore compare models by the amount of information they provide about the described systems. If the characterization of a system in model A can be obtained from its characterization in model B , then model B contains at least as much information about the system as model A . If A describes the properties of interest, then B could be considered to contain unnecessary details. Back and Mannila ([BM 85]) discuss these ideas in a general setting, and our treatment is inspired by theirs.

In our framework, we are particularly interested in models of I/O-systems that are compositional with respect to the composition and abstraction operations. We shall therefore in this section present a few such models, and compare them to each other. The models differ in whether they describe communication events, safety properties, or liveness properties of a system. We compare the models to each other with respect to information-content. We also address the question how the models relate to other potential models.

3.4.1 Preliminaries

In this section, we assume a countably infinite set $\mathcal{E}$ of communication events. We assume that $\mathcal{E}$ is partitioned into an infinite number of non-empty subsets. Let Γ be the set of I/O-systems whose communication events are contained in finitely many subsets of $\mathcal{E}$ and that have at most countably many states. Let Op_Γ be the composition and abstraction operations on the set of I/O-systems Γ .

The motivation for partitioning $\mathcal{E}$ into an infinite number of subsets is that we wish to exclude systems with all events in $\mathcal{E}$ as communication events. Intuitively, each subset of $\mathcal{E}$ can be regarded as the set of communication events on a certain channel. The requirement on the systems in Γ then states that a system can communicate only via a finite number of channels.

Definition 3.4.1. A *model* $\mathcal{M}$ of Γ is a set $\mathcal{D}_\mathcal{M}$ and a function $h_\mathcal{M} : \Gamma \mapsto \mathcal{D}_\mathcal{M}$. A model $\mathcal{M}$ of Γ is *compositional* with respect to the operations Op_Γ if for each (partial) operation $op_\Gamma \in Op_\Gamma$ there is a (partial) operation $op_\mathcal{M}$ on $\mathcal{D}_\mathcal{M}$ such that

$$h_\mathcal{M}(op_\Gamma(a_1, \dots, a_n)) = op_\mathcal{M}(h_\mathcal{M}(a_1), \dots, h_\mathcal{M}(a_n))$$

for any $a_1, \dots, a_n$ in Γ , such that $op_\Gamma(a_1, \dots, a_n)$ is defined.

□

Examples of models of Γ that are compositional with respect to Op_Γ are the model defined in section 3.3, and Γ itself.

We shall now define how two models are compared according to how much information they contain about the objects in Γ . Intuitively, $\mathcal{M}$ contains less information than $\mathcal{M}'$ if the description $h_{\mathcal{M}'}(a)$ of an object $a \in \Gamma$ in $\mathcal{M}'$ contains enough information to find the description $h_{\mathcal{M}}(a)$ of a in $\mathcal{M}$.

Definition 3.4.2. A model $\mathcal{M}$ (of Γ) is *more abstract* than another model $\mathcal{M}'$, denoted $\mathcal{M} \sqsubseteq \mathcal{M}'$ if $h_{\mathcal{M}'}(a) = h_{\mathcal{M}'}(b)$ implies $h_{\mathcal{M}}(a) = h_{\mathcal{M}}(b)$ for all $a, b \in \Gamma$. $\square$

If $\mathcal{M} \sqsubseteq \mathcal{M}'$, then the description in $\mathcal{M}$ of the object $a \in \Gamma$ can be obtained from the description of a in $\mathcal{M}'$.

Notation.

$\mathcal{M} \cong \mathcal{M}'$ denotes that $\mathcal{M} \sqsubseteq \mathcal{M}' \sqsubseteq \mathcal{M}$

$\mathcal{M} \sqsubset \mathcal{M}'$ denotes that $\mathcal{M} \sqsubseteq \mathcal{M}' \not\sqsubseteq \mathcal{M}$

Lemma 3.4.3. Let $\mathcal{M}$ and $\mathcal{M}'$ be models of Γ . Then $\mathcal{M} \sqsubset \mathcal{M}'$ if and only if

- (i) $h_{\mathcal{M}'}(a) = h_{\mathcal{M}'}(b)$ implies $h_{\mathcal{M}}(a) = h_{\mathcal{M}}(b)$ for all $a, b \in \Gamma$, and
- (ii) there are $c, d \in \Gamma$ such that $h_{\mathcal{M}'}(c) \neq h_{\mathcal{M}'}(d)$ and $h_{\mathcal{M}}(c) = h_{\mathcal{M}}(d)$.

Proof. Immediate from definition 3.4.2. $\square$

3.4.2 Comparison between models of I/O-systems

In this subsection, we define three different models of I/O-systems, and relate them to each other using the concepts developed above.

Definition 3.4.4. For an I/O-system $N \in \Gamma$, let the triple $\langle I(N), O(N), Q(N) \rangle$ be the denotation $\llbracket N \rrbracket$ of N as in definition 3.3.1.

- The *event model* $\mathcal{M}_1$ maps N to the tuple $\langle I(N), O(N) \rangle$.
- The *safety model* $\mathcal{M}_2$ maps N to the triple $\langle I(N), O(N), T(N) \rangle$, where T_N is the set of finite prefixes of sequences in Q_N .
- The *liveness model* $\mathcal{M}_3$ maps a network N to its denotation $\llbracket N \rrbracket$.

$\square$

Intuitively, the event model maps a system to the set of communication events of a network. The safety model in addition describes which finite sequences of events may occur in finite partial computations of a system. The liveness model also characterizes the finite sequences that must always be extended by the system, and characterizes restrictions on infinite sequences.

Theorem 3.4.5. The event model, the safety model, and the liveness model are compositional with respect to the operations Op_r .

Proof. For the liveness model, this is stated by theorem 3.3.2. For the other two models, the following operations correspond to composition and abstraction. We use N_{comp} to denote $N_1 \parallel \dots \parallel N_k$.

Event Model: For composition the operations are

$$O(N_1 \parallel \dots \parallel N_k) = \bigcup_{i=1}^k O(N_i)$$

$$I(N_1 \parallel \dots \parallel N_k) = \bigcup_{i=1}^k I(N_i) - O(N)$$

and for abstraction

$$I(N \setminus E) = I(N)$$

$$O(N \setminus E) = O(N) - E$$

Safety Model: The operations for composition and abstraction of input and output events are as for the event model. For the set of traces, the operations are the following:

$$T(N_1 \parallel \dots \parallel N_k) = \{q \in (E(N_{comp}))^* \mid \text{for all } i \quad q \upharpoonright (E(N_i)) \in T(N_i)\}$$

$$T(N \setminus E) = \{q \setminus E \mid q \in T(N)\}$$

where we use $E(N)$ to denote $I(N) \cup O(N)$ for an I/O-system N . The operations for input and output events follow directly from theorem 3.3.2. The rule for composition and abstraction of sets of traces uses a proof similar to the proof of theorem 3.3.2. $\square$

The main theorem of this section is a comparison between these models and a result about their relationship to other possible compositional models of I/O-systems.

Theorem 3.4.6. Let the models $\mathcal{M}_1, \mathcal{M}_2$, and $\mathcal{M}_3$ be as in definition 3.4.4.

- (i) $\mathcal{M}_1 \sqsubset \mathcal{M}_2 \sqsubset \mathcal{M}_3$
- (ii) There is no model $\mathcal{M}$ of Γ , which is compositional with respect to the operations in Op_r for which

$$\mathcal{M}_1 \sqsubset \mathcal{M} \sqsubset \mathcal{M}_2$$

- (iii) There is no model $\mathcal{M}$ of Γ , which is compositional with respect to the composition and abstraction operations for which

$$\mathcal{M}_2 \sqsubset \mathcal{M} \sqsubset \mathcal{M}_3$$

Proof. The proof of (i) follows immediate from definition 3.4.4. For the proof of (ii) and (iii) we need the following two lemmas, whose proofs are given in section 3.6.

Lemma 3.4.7. Let the models $\mathcal{M}_1$ and $\mathcal{M}_2$ be as in definition 3.4.4. Let a, b, c , and d be I/O-systems in Γ for which $h_{\mathcal{M}_1}(a) = h_{\mathcal{M}_1}(b)$ and $h_{\mathcal{M}_1}(c) = h_{\mathcal{M}_1}(d)$. Assume that

- (i) there is a trace $t \in (I(a) \cup O(a))^*$ such that $t \notin T(a)$ but $t \in T(b)$.
- (ii) $T(c) \subseteq T(d)$.

Then there is an expression (or context) $\Phi[\cdot]$, built from systems in Γ and operations in Op_Γ , such that

$$\begin{aligned} h_{\mathcal{M}_2}(\Phi[a]) &= h_{\mathcal{M}_2}(c) \\ h_{\mathcal{M}_2}(\Phi[b]) &= h_{\mathcal{M}_2}(d) \end{aligned}$$

□

Lemma 3.4.8. Let the models $\mathcal{M}_2$ and $\mathcal{M}_3$ be as in definition 3.4.4. Let a, b, c , and d be I/O-systems in Γ for which $h_{\mathcal{M}_2}(a) = h_{\mathcal{M}_2}(b)$ and $h_{\mathcal{M}_2}(c) = h_{\mathcal{M}_2}(d)$. Assume that

- (i) there is a finite or infinite sequence $q \in (I(a) \cup O(a))^\dagger$ such that $q \notin Q(a)$ but $q \in Q(b)$.
- (ii) $Q(c) \subseteq Q(d)$.

Then there exists a context $\Phi[\cdot]$ expressible with composition and abstraction of systems in Γ , such that

$$\begin{aligned} h_{\mathcal{M}_3}(\Phi[a]) &= h_{\mathcal{M}_3}(c) \\ h_{\mathcal{M}_3}(\Phi[b]) &= h_{\mathcal{M}_3}(d) \end{aligned}$$

□

We can now proceed to the proofs of cases (ii) and (iii) of theorem 3.4.6.

(ii) Assume that there exists a model $\mathcal{M}$ such that $\mathcal{M}_1 \sqsubset \mathcal{M} \sqsubset \mathcal{M}_2$. We shall derive a contradiction. By $\mathcal{M} \sqsubset \mathcal{M}_2$ and lemma 3.4.3, there are I/O-systems $a, b \in \Gamma$ such that

$$h_{\mathcal{M}}(a) = h_{\mathcal{M}}(b) \quad \text{and} \quad h_{\mathcal{M}_2}(a) \neq h_{\mathcal{M}_2}(b) \quad (1)$$

By $\mathcal{M}_1 \sqsubset \mathcal{M}$ and lemma 3.4.3, there are I/O-systems $c, d \in \Gamma$ such that

$$h_{\mathcal{M}_1}(c) = h_{\mathcal{M}_1}(d) \quad \text{and} \quad h_{\mathcal{M}}(c) \neq h_{\mathcal{M}}(d) \quad (2)$$

We shall first assume that $T(c) \subseteq T(d)$. We then use lemma 3.4.7 to conclude that there is an expression (or context) $\Phi[\cdot]$, built from systems in Γ and operations in Op_Γ , such that

$$h_{\mathcal{M}_2}(\Phi[a]) = h_{\mathcal{M}_2}(c) \quad (3)$$

$$h_{\mathcal{M}_2}(\Phi[b]) = h_{\mathcal{M}_2}(d) \quad (4)$$

If $\mathcal{M}$ is compositional with respect to the operations in Op_Γ , it must be compositional with respect to the context $\Phi[\cdot]$. By (1) we therefore get $h_{\mathcal{M}}(\Phi[a]) = h_{\mathcal{M}}(\Phi[b])$.

By (3) and $\mathcal{M} \sqsubseteq \mathcal{M}_2$ we get $h_{\mathcal{M}}(\Phi[a]) = h_{\mathcal{M}}(c)$.

By (4) and $\mathcal{M} \sqsubseteq \mathcal{M}_2$ we get $h_{\mathcal{M}}(\Phi[b]) = h_{\mathcal{M}}(d)$.

This gives $h_{\mathcal{M}}(c) = h_{\mathcal{M}}(\Phi[a]) = h_{\mathcal{M}}(\Phi[b]) = h_{\mathcal{M}}(d)$, contradicting (2).

We shall finally consider the case in which $T(c) \subseteq T(d)$ does not hold. We then replace either c or d by another system in Γ . Namely, consider the I/O-system u for which $h_{\mathcal{M}_1}(c) = h_{\mathcal{M}_1}(u)$ and $T(u) = (I(u) \cup O(u))^*$, i.e., u has the same input and output events as c , but may perform any sequence of communication events. The system u can be constructed as an I/O-system which can perform any communication event at any moment. The point of bringing in u is that $T(c) \subseteq T(u)$ and $T(d) \subseteq T(u)$. Therefore, by (2) it follows that $h_{\mathcal{M}}(u)$ is different from either $h_{\mathcal{M}}(c)$ or $h_{\mathcal{M}}(d)$. By taking d or c as u , we may therefore without loss of generality assume that $T(c) \subseteq T(d)$.

(iii) The proof in this case is analogous to the proof in the preceding case, using $\mathcal{M}_3$ for $\mathcal{M}_2$ and $\mathcal{M}_2$ for $\mathcal{M}_1$. Instead of using lemma 3.4.7 to derive (3) and (4), we use lemma 3.4.8.

To cover the case when $Q(c) \subseteq Q(d)$ does not hold, we use the system u for which $T(u) = (I(u) \cup O(u))^\dagger$, i.e., u has the same input and output events as c , but may perform any finite or infinite sequence of communication events as a quiescent trace. The system u can be constructed as an I/O-system which can perform any communication event at any moment, and become quiescent (not produce any more output events) at any moment.

□

Theorem 3.4.7 states that there is no model of Γ which is compositional with respect to the composition and abstraction operations, contains all information about input events and output events, but contains information about only some of the safety properties. The theorem also states that there is no compositional model which contains all information about safety properties that $\mathcal{M}_2$ contains, but only information about some of the liveness properties described by the liveness model.

This does not mean that there are no other compositional models of Γ . The model $\mathcal{M}_4$, defined by $h_{\mathcal{M}_4}(N) = \langle I(N), O(N), T(N), U(N) \rangle$, where the components $I(N), O(N), T(N)$ are as in the safety model, and $U(N)$ is the set of sequences of communication events in a *finite* quiescent computation, is compositional, for we have the rules

$$U(N_1 \parallel \dots \parallel N_k) = \{q \in (E(N_1 \parallel \dots \parallel N_k))^* \mid \text{for all } i \ q[(E(N_i)) \in U(N_i)]\}$$

$$U(N \setminus E) = \{q \setminus E \mid q \in U(N)\}$$

which can be proven similarly to the rules in theorem 3.3.2.

The relationship between $\mathcal{M}_4$ and the previous models is that $\mathcal{M}_2 \subseteq \mathcal{M}_4$, but $\mathcal{M}_4 \not\subseteq \mathcal{M}_3$ and $\mathcal{M}_3 \not\subseteq \mathcal{M}_4$. The reason for why $\mathcal{M}_4 \not\subseteq \mathcal{M}_3$ is that $\mathcal{M}_3$ does not contain any information whether a finite quiescent trace occurs in a finite or in an infinite computation. The reason for why $\mathcal{M}_3 \not\subseteq \mathcal{M}_4$ is that $\mathcal{M}_4$ cannot distinguish between systems with different fairness requirements.

In our earlier work ([Jo 85]) we defined a compositional model, $\mathcal{M}_5$, such that $\mathcal{M}_3 \subseteq \mathcal{M}_5$ and $\mathcal{M}_4 \subseteq \mathcal{M}_5$ is defined. $\mathcal{M}_5$ extends $\mathcal{M}_3$ by adding information that for each finite quiescent trace tells whether it arises from a finite or an infinite computation.

3.5 Applications to Kahn-networks.

An important class of I/O-systems is the class of Kahn-networks. A Kahn-network consists of a set of processes that communicate over unbounded FIFO channels.

For Kahn-networks with only deterministic processes, Kahn ([Ka 74]) has presented an elegant model where a Kahn-network is characterized by a function from histories of input channels to histories of output channels. Unfortunately, this model does not generalize directly to nondeterministic networks. The obvious generalization of characterizing a network by a relation between the histories of the channels of the network is not compositional, as has been shown by Brock and Ackermann ([BA 81]). Several models of non-deterministic Kahn-networks have been proposed (e.g., [BA 81], [Broy 83], [BM 82], [Ko 78], [KP 84], [Pa 83], [Pr 82], [SN 85]).

A Kahn-network can be described as an I/O-system by including the FIFO channels over which it communicates with its environment. We can then apply the results of section 3.3 to obtain a compositional model of nondeterministic Kahn-networks.

In this section, we relate the history model for Kahn-networks to the model derived from the quiescent-traces-model. The main result is that any compositional model which extends the history model contains more information than the quiescent-traces-model. Thus the quiescent-traces model is the minimal extension (in the sense of containing least information) of the history model for nondeterministic Kahn-networks.

3.5.1 Kahn-networks as I/O-systems

To formalize Kahn-networks, we assume a countable set of *channel names*, and for each channel name a countable set of *messages* that may be transmitted over that channel. A *communication event* represents the transmission of a message over a channel and is written $chan(m)$ where $chan$ is a channel name, and m is a message that may be transmitted over $chan$.

A (non-deterministic) *Kahn-network* is a system which communicates with its environment via FIFO channels. A Kahn-network has a finite set of *input channels* and a finite set of *output channels*, which are disjoint from the input channels. A Kahn-network is a tuple $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$, where

I is the set of all communication events of form $chan_i(m)$ where m is a message that can be transmitted over the input channel $chan_i$.

O is the set of all communication events of form $chan_i(m)$ where m is a message that can be transmitted over the output channel $chan_i$.

Σ consists of tuples $\langle \sigma_{int}, seq_1, \dots, seq_k \rangle$ where

σ_{int} belongs to a set of *internal states*

seq_i is a sequence of messages that may be transmitted over the channel $chan_i$ of N , i.e., seq_i ranges over finite sequences of messages that may be transmitted over $chan_i$.

$\sigma^0 = \langle \sigma_{int}^0, \langle \rangle, \dots, \langle \rangle \rangle$, with all channels empty.

R is a subset of $\Sigma \times (I \cup O \cup \{\tau\}) \times \Sigma$. Each element in R is a transition of one of the following forms.

- 1) Transitions that only concern the internal component σ_{int} of σ . These are silent transitions of the form

$$\langle \sigma_{int}, seq_1, \dots, seq_k \rangle \xrightarrow{\tau} \langle \sigma'_{int}, seq_1, \dots, seq_k \rangle$$

- 2) Transitions which receive a message into an input channel $chan_i$. These do not change the internal state σ_{int} .

$$\langle \sigma_{int}, \dots, seq_i, \dots \rangle \xrightarrow{chan_i(m)} \langle \sigma_{int}, \dots, seq_i \bullet m, \dots \rangle$$

- 3) Transitions which read a message from an input channel $chan_i$, and may change the internal state.

$$\langle \sigma_{int}, \dots, m \bullet seq_i, \dots \rangle \xrightarrow{\tau} \langle \sigma'_{int}, \dots, seq_i, \dots \rangle$$

- 4) Transitions which put a message into an output channel $chan_i$.

$$\langle \sigma_{int}, \dots, seq_i, \dots \rangle \xrightarrow{\tau} \langle \sigma'_{int}, \dots, seq_i \bullet m, \dots \rangle$$

- 5) Transitions that transmit a message from an output channel $chan_i$.

$$\langle \sigma_{int}, \dots, m \bullet seq_i, \dots \rangle \xrightarrow{chan_i(m)} \langle \sigma_{int}, \dots, seq_i, \dots \rangle$$

$\mathcal{F}$ is a collection of fairness sets, subject to the following constraints:

- (i) All transitions of form 1), 3), 4), and 5) are members of some fairness set.
- (ii) For each output channel $chan_i$, the corresponding transitions in R of type 5) constitute one fairness set. In other words, the output channels of a Kahn-network eventually transmit each message that is put into the channel.

It is not difficult to verify that a Kahn-network is an I/O-system, according to definition 3.2.1. The composition and abstraction operations on Kahn-networks is defined in the same way as for I/O-systems.

3.5.2 Comparison with history model

The liveness model $\mathcal{M}_3$ is defined on the class of Kahn-networks in the same way as for I/O-systems. In order to compare it with the history model, we define the history model for Kahn-networks.

Definition 3.5.1. Let N be a Kahn-network with channels $chan_1, \dots, chan_k$.

- A *history tuple* of N is a tuple $\langle seq_1, \dots, seq_k \rangle$ such that there is a computation of N in which seq_i is the (finite or infinite) sequence of communication events $chan_i(m)$ on the channel $chan_i$.
- The *history model* $\mathcal{M}_H$ is defined by

$$h_{\mathcal{M}_H}(N) = \langle I(N), O(N), H(N) \rangle$$

where $H(N)$ is the set of history tuples of N .

□

We now state the main theorem of this section, which states that the liveness model is a minimal compositional extension of the history model, in the sense of containing least information about Kahn-networks.

Theorem 3.5.2. Let Υ be the set of Kahn-networks. Let $\mathcal{M}_H$ be the history model, and $\mathcal{M}_3$ the liveness model on Υ . Then

- (i) $\mathcal{M}_H \subseteq \mathcal{M}_3$
- (ii) If $\mathcal{M}$ is a model of Υ , compositional with respect to composition and abstraction, such that $\mathcal{M}_H \subseteq \mathcal{M}$, then $\mathcal{M}_3 \subseteq \mathcal{M}_H$.

Proof. The proof of (i) follows immediate from the definition of $\mathcal{M}_H$.

For the proof of (ii) we need the following lemma, whose proof is given in section 3.6.

Lemma 3.5.3. Assume that there are two Kahn-networks $a, b \in \mathcal{T}$ such that $h_{\mathcal{M}_1}(a) = h_{\mathcal{M}_1}(b)$, and that there is a sequence of events q such that $q \notin Q(a)$ and $q \in Q(b)$. Then there is an expression (or context) $\Phi[\cdot]$ built from Kahn-networks, and the composition and abstraction operations, such that

$$h_{\mathcal{M}_H}(\Phi[a]) \neq h_{\mathcal{M}_H}(\Phi[b]) \quad (2)$$

□

We can now prove case (ii) as follows: Assume that $\mathcal{M}$ is a compositional model, such that $\mathcal{M}_H \sqsubseteq \mathcal{M}$. This means that $h_{\mathcal{M}}(a) = h_{\mathcal{M}}(b)$ implies $h_{\mathcal{M}_H}(a) = h_{\mathcal{M}_H}(b)$ for all Kahn-networks $a, b \in \mathcal{T}$. In the proof we shall assume that $\mathcal{M}_3 \not\sqsubseteq \mathcal{M}$ is false, and derive a contradiction.

By $\mathcal{M}_3 \not\sqsubseteq \mathcal{M}$ and definition 3.4.2, there are Kahn-networks $a, b \in \mathcal{T}$ such that

$$h_{\mathcal{M}}(a) = h_{\mathcal{M}}(b) \quad \text{and} \quad h_{\mathcal{M}_3}(a) \neq h_{\mathcal{M}_3}(b) \quad (1)$$

Without loss of generality, we can assume that there is a sequence q such that $q \notin Q(a)$ and $q \in Q(b)$. By lemma 3.5.2, there is an expression (or context) $\Phi[\cdot]$ built from Kahn-networks, and the composition and abstraction operations, such that

$$h_{\mathcal{M}_H}(\Phi[a]) \neq h_{\mathcal{M}_H}(\Phi[b]) \quad (2)$$

If $\mathcal{M}$ is compositional with respect to composition and abstraction, it must be compositional with respect to the context $\Phi[\cdot]$. By (1) we therefore get

$$h_{\mathcal{M}}(\Phi[a]) = h_{\mathcal{M}}(\Phi[b]) \quad (3)$$

But by $\mathcal{M}_H \sqsubseteq \mathcal{M}$ this implies $h_{\mathcal{M}_H}(\Phi[a]) = h_{\mathcal{M}_H}(\Phi[b])$, which contradicts (2).

□

3.6 Proofs from Chapter 3

Proof of proposition 3.2.7.

We must prove that the composition and abstraction operations result in I/O-systems if their arguments are I/O-systems.

Composition. Let $N = N_1 \parallel \dots \parallel N_k$, where we assume that N_i is the I/O-system $\langle I_i, O_i, \Sigma_i, \sigma_i^0, R_i, \mathcal{F}_i \rangle$, and $N = \langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$. We verify that N is an I/O-system, by checking the three conditions in definition 3.2.1.

- 1) Assume that $e \in I$ is an input event of N , and that $\sigma = \langle \sigma_1, \dots, \sigma_k \rangle \in \Sigma$ is a state of N . Since $I = (\bigcup_{i=1}^k I_i) \setminus O$, it follows that e is an input event of one or several N_i , and not an output event for any N_i . If e is an input event of N_i , there is a transition $\sigma_i \xrightarrow{e} \sigma'_i \in R_i$. It follows that there is a transition from σ labeled by e , obtained from such a transition for each N_i such that $e \in I_i$, namely

$$\langle \sigma_1, \dots, \sigma_{i_1}, \dots, \sigma_{i_l}, \dots, \sigma_k \rangle \xrightarrow{e} \langle \sigma_1, \dots, \sigma'_{i_1}, \dots, \sigma'_{i_l}, \dots, \sigma_k \rangle$$

- 2) Assume that $\sigma \xrightarrow{e} \sigma' \in R$ with $e \in I$. We must show that no fairness set in $\mathcal{F}$ contains this transition. As in case 1) it follows that $\sigma \xrightarrow{e} \sigma'$ is derived from only input transitions in $R_1, \dots, R_k$. Neither of these transitions is in any fairness set of $\mathcal{F}_1, \dots, \mathcal{F}_k$, and hence $\sigma \xrightarrow{e} \sigma'$ is not in any fairness set of $\mathcal{F}$.
- 3) Assume that $\sigma \xrightarrow{e} \sigma' \in R$ and that $E \in O \cup \{\tau\}$. Since $O = \bigcup_{i=1}^k O_i$, it follows that $\sigma \xrightarrow{e} \sigma'$ is derived either from a transition labeled by τ , or from a transition in some R_j labeled by an output event (and possibly other transitions). In either case, $\sigma \xrightarrow{e} \sigma'$ is derived from a transition which is in some fairness set in $\mathcal{F}_1, \dots, \mathcal{F}_k$, and hence it is in some fairness set in $\mathcal{F}$.

Abstraction We verify that $N \setminus E$ is an I/O-system if N is an I/O-system and e is a subset of the output events of N , by checking the conditions in definition 3.2.1.

- 1) Assume that $e \in I$ is an input event of $N \setminus E$, and that $\sigma \in \Sigma$ is a state of $N \setminus E$. Since N is an I/O-system, it follows that $\sigma \xrightarrow{e} \sigma' \in R$, and hence by the definition of abstraction that $\sigma \xrightarrow{e} \sigma' \in R \setminus E$.
- 2) Assume that $\sigma \xrightarrow{e} \sigma' \in R \setminus E$ with $e \in I$ being an input event of $N \setminus E$. It follows that $\sigma \xrightarrow{e} \sigma' \in R$, and that this transition is not member of any fairness set in $\mathcal{F}$. Hence it is not in any fairness set of $\mathcal{F}_E$.
- 3) Assume that $\sigma \xrightarrow{e} \sigma' \in R \setminus E$ and that $e \in ((O - E) \cup \{\tau\})$. This transition is derived from a transition $\sigma \xrightarrow{e'} \sigma' \in R$ with $e \in (O \cup \{\tau\})$, which is in some fairness set of $\mathcal{F}$. Hence $\sigma \xrightarrow{e} \sigma'$ is in some fairness set of $\mathcal{F}_E$. $\square$

Proof of theorem 3.3.2

We shall prove that the rules for composition and abstraction of sets of quiescent traces follow from the definitions of these operations on I/O-systems.

Composition: Let $N = N_1 \parallel \dots \parallel N_k$, where we assume that N_i is the I/O-system $\langle I_i, O_i, \Sigma_i, \sigma_i^0, R_i, \mathcal{F}_i \rangle$, and $N = \langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$. Let $E(N) = I(N) \cup O(N)$ and $E(N_i) = I(N_i) \cup O(N_i)$. We must prove the following rules:

$$O(N) = \bigcup_{i=1}^k O(N_i)$$

$$I(N) = \bigcup_{i=1}^k I(N_i) - O(N)$$

$$Q(N) = \{q \in (E(N))^\dagger \mid q[E(N_i)] \in Q(N_i) \text{ for all } i\}$$

The rules for composition of input and output events follow immediately from definition 3.2.5. We shall prove the third rule in two parts

First step: Let $q \in (E(N))^\dagger$. We prove that if $q \in Q(N)$, then $q[E(N_i)] \in Q(N_i)$ for each i . Assume that q is the sequence of communication events in a computation C of N of the form

$$C = \langle \sigma_1^0, \dots, \sigma_k^0 \rangle \xrightarrow{e^1} \langle \sigma_1^1, \dots, \sigma_k^1 \rangle \xrightarrow{e^2} \dots \xrightarrow{e^n} \langle \sigma_1^n, \dots, \sigma_k^n \rangle \xrightarrow{e^{n+1}} \dots$$

We shall construct a computation C_i of N_i in which the sequence of communication events is $q[E(N_i)]$. The computation C_i is constructed by replacing each transition

$$\langle \sigma_1^{n-1}, \dots, \sigma_i^{n-1}, \dots, \sigma_k^{n-1} \rangle \xrightarrow{e^n} \langle \sigma_1^n, \dots, \sigma_i^n, \dots, \sigma_k^n \rangle$$

in C by $\sigma_i^{n-1} \xrightarrow{e^n} \sigma_i^n$ if this transition is in R_i , and by the single state σ_i^{n-1} otherwise (in the latter case we must have $\sigma_i^{n-1} = \sigma_i^n$).

The result of this transformation is by construction a partial computation C_i of N_i . The sequence of communication events in C_i is $q[E(N_i)]$, since a transition labeled by a communication event e^n is collapsed to a state if and only if $e^n \notin E(N_i)$. It remains to prove that C_i is indeed a computation of N_i . The conditions for being a computation are checked below.

Initialization. σ_i^0 is the initial state of N_i , since it is the i th component of the initial state of N .

State-to-state sequencing. This condition follows from the transformation from C to C_i and definition 3.2.5.

Fairness. If C_i is infinite, then C is infinite. Since each fairness set of N_i generates a corresponding fairness set of N , and each output or silent transition in C_i has a corresponding transition in C , it follows that C_i is fair if C is fair.

Quiescence. Assume that C_i is finite, and ends in the state σ_i^f . We must consider two cases:

- (i) C is finite. It must then end in a state $\langle \sigma_1^f, \dots, \sigma_i^f, \dots, \sigma_k^f \rangle$ which is quiescent. It follows that only input transitions are enabled in σ_i^f .
- (ii) C is infinite. C must then have an infinite suffix C^{suff} whose states have σ_i^f as the i th component. Assume that an output or silent transition in R_i is enabled in σ_i^f . The fairness sets in $\mathcal{F}_i$ contain exactly all output and silent transitions in R_i , and hence a subcollection $\mathcal{F}^{sub}$ of $\mathcal{F}$ contains exactly all transitions derived from an output or silent transition in R_i . Since all states in C^{suff} have σ_i^f as the i th component, it follows that a transition in a fairness set of $\mathcal{F}^{sub}$ is enabled throughout C^{suff} . However, no transition from $\mathcal{F}^{sub}$ is ever taken in C^{suff} . Hence C does not satisfy the requirement of fairness, leading to a contradiction.

Second part: Before proving the second part, we make some preliminary definitions and state a lemma. We shall use the notation $\Xi(C)$ to denote the sequence of communication events in a sequence of transitions C .

In this proof, we represent each transition of N

$$\langle \sigma_1, \dots, \sigma_{i_1}, \dots, \sigma_{i_m}, \dots, \sigma_k \rangle \xrightarrow{e} \langle \sigma_1, \dots, \sigma'_{i_1}, \dots, \sigma'_{i_m}, \dots, \sigma_k \rangle$$

by the set of transitions of components N_i from which the transition is derived according to definition 3.2.5. That is, if the above transition can be derived from the transitions

$$\sigma_{i_1} \xrightarrow{e} \sigma'_{i_1} \quad , \quad \dots \quad , \quad \sigma_{i_m} \xrightarrow{e} \sigma'_{i_m}$$

of the components $N_{i_1}, \dots, N_{i_m}$, according to definition 3.2.5, then the set

$$\{\sigma_{i_1} \xrightarrow{e} \sigma'_{i_1} \quad , \quad \dots \quad , \quad \sigma_{i_m} \xrightarrow{e} \sigma'_{i_m}\}.$$

is a *set-representation* of the above transition. A set-representation of a sequence of transitions C is a sequence of set-representations of the transitions in C .

The main ingredient of the proof is the following lemma, which will be proven after the proof of this theorem.

Lemma 3.6.1. Let $C_1, \dots, C_k$ be partial computations of $N_1, \dots, N_k$. Let q be a sequence of communication events of $N_1 \parallel \dots \parallel N_k$, such that $q \upharpoonright E(N_i) = \Xi(C_i)$. Then there is a partial computation C of N , such that $\Xi(C) = q$, and such that there is a set-representation of C in which the sequence of transitions of each N_i is the sequence of transitions in C_i . $\square$

We now proceed with the second part of the proof. We shall prove that if $q \upharpoonright E(N_i) \in Q(N_i)$ for each i , then $q \in Q(N)$.

Since $q[E(N_i) \in Q(N_i)]$, there is a computation C_i of N_i such that $\Xi(C_i) = q[E(N_i)]$. We shall show that there is a computation C of N such that $\Xi(C) = q$. By lemma 3.6.1, there is a partial computation C of N , such that $\Xi(C) = q$, and such that there is a set-representation of C in which the sequence of transitions of each N_i is the sequence of transitions in C_i . It only remains to verify that C is really a computation. We check the two last requirements in definition 3.2.3.

- *Quiescence.* Assume that C is finite, and ends in the state $\sigma^f = \langle \sigma_1^f, \dots, \sigma_k^f \rangle$. The last transition of each N_i in the set-representation of C is also the last transition of C_i , which leads to the state σ_i^f . Since C_i is a computation, there is no transition from σ_i^f labeled by an event in $O(N_i) \cup \{\tau\}$. By definition 3.2.5 there is no transition labeled by an event in $O(N) \cup \{\tau\}$ from σ^f .
- *Fairness.* Assume that F is a fairness set of N , which is the set of transitions obtained from a fairness set F_i of some N_i . Assume that C has only finitely many transitions from F . By the lemma, C_i is the sequence of transitions of N_i in a set-representation of C . Hence C_i contains only finitely many transitions from F_i . If C_i is infinite, then no transition from F_i is enabled beyond a certain point in C_i . If C_i is finite, no output or silent transition of R_i and hence of F_i is enabled in its last state. In both cases, no transition from F is enabled beyond a certain point in C . Hence a transition from F is only enabled for finitely many states, and C satisfies the fairness requirement.

Abstraction:

From the definition of abstraction of I/O-systems it follows that

$$\sigma^0 \xrightarrow{e^1} \sigma^1 \xrightarrow{e^2} \dots \xrightarrow{e^n} \sigma^n \xrightarrow{e^{n+1}} \dots$$

is a computation of N if and only if

$$\sigma^0 \xrightarrow{e^{1'}} \sigma^1 \xrightarrow{e^{2'}} \dots \xrightarrow{e^{n'}} \sigma^n \xrightarrow{e^{n+1}'} \dots$$

is a computation of $N \setminus E$, where $e^{n'} = \tau$ if $e^n \in E$, otherwise $e^{n'} = e^n$. It follows that q is a sequence of communication events in a computation of N iff $q \setminus E$ is a sequence of communication events in a computation of $N \setminus E$.

□

Lemma 3.6.1 states that the sequences of transitions C_i can be “merged” into a sequence C under certain conditions. Before proving lemma 3.6.1, we need some preliminaries.

We first review some standard definitions and results. Let A be a set. A *preorder* on A is a binary relation which is reflexive and transitive. A *partial order* on A

is a preorder which is also antisymmetric. A preorder $\leq$ on A generates a set of equivalence classes of A , for which $a, b \in A$ are in the same equivalence class if and only if $a \leq b \leq a$. The preorder $\leq$ induces a relation $\preceq$ between equivalence classes: An equivalence class containing a is related by $\preceq$ to the equivalence class containing b if and only if $a \leq b$. The relation $\preceq$ is a partial order on the set of equivalence classes.

If A is a set with a partial order $\preceq$, call $x \in A$ *finitely preceded* if there are only finitely many $y \in A$ such that $y \preceq x$.

Lemma 3.6.2. (Linearization Lemma) Let A be a set with a partial order $\preceq$. If A has at most countably many elements, and each element of A is finitely preceded, then the elements of A can be ordered into a linear sequence, which respects $\preceq$.

Proof. In the proof we shall construct a linear sequence $a_1 a_2 a_3 \dots$ by adding one element at a time.

First assign a unique positive integer $\rho(x)$ to each $x \in A$. This is possible, since A has at most countably many elements. Next we construct the sequence *seq* of elements of A by the following procedure.

```

seq := ⟨⟩
while not all elements of A in seq do
  seq := seq • x where
    ρ(x) = min{ρ(x') | x' ∉ seq and (∀y ≠ x')[y ⪯ x' ⊃ y ∈ seq]}
od

```

The sequence constructed by this procedure is an extension of $\preceq$, since an element is added only if all $y \prec x$ are already in *seq*. Furthermore, each $x \in A$ will eventually be put into the sequence, because if all $y \prec x$ are in *seq* then x will eventually become the element with the lowest $\rho(x)$ which is not in *seq*. Since each x is finitely preceded, we can by induction conclude that each $x \in A$ eventually gets into *seq*. $\square$

We now proceed with the proof of lemma 3.6.1. Let $\mathfrak{S}$ be the set of *occurrences* of component transitions in $C_1, \dots, C_k$, i.e., we distinguish between different occurrences of the same transition in a computation C_i . Define a preorder $\leq$ on $\mathfrak{S}$ as the transitive closure of

- (i) $t_i^n \leq t_i^{n+1}$ if t_i^n is the n th transition of C_i , and t_i^{n+1} is the $(n+1)$ st transition of C_i .
- (ii) $t_i \leq t_j$ if t_i is a non-silent transition of C_i which corresponds to the n th communication event in q , and t_j is a non-silent transition of C_j which corresponds to the n th or $(n+1)$ st communication event in q .

The preorder $\leq$ induces a set of equivalence classes on $\mathfrak{S}$. Let $\preceq$ be the induced partial order between these equivalence classes. We claim that

- (1) Each equivalence class generated by $\leq$ is a set-representation of a transition of N .
- (2) Each equivalence class is finitely preceded.

The conclusion of the lemma follows from these claims as follows: According to the linearization lemma there exists a sequence of equivalence classes which respects the ordering requirements (i) and (ii). By claim (1) and the definition of $\leq$, this sequence is the set-representation of a partial computation C of N . Now requirement (i) ensures that the transitions of N_i occur in the same order as in C_i , and requirement (ii) ensures that $\Xi(C) = q$.

It remains to prove the claims (1) and (2). We make the following observation: Assume that $t_i \leq t_j$ for occurrences of transitions t_i in C_i and t_j in C_j with $i \neq j$. Then there must be integers n_i and n_j with n_i less than or equal to n_j , such that t_i precedes a transition in C_i which corresponds to the n_i th communication event of q , and t_j follows a transition in C_j which corresponds to the n_j th communication event of q . Claim (1) now follows, since $t_i \leq t_j \leq t_i$ implies that $n_i = n_j$ and that t_i and t_j correspond to the same communication event in q . Hence they are in the same transition of N . To verify claim (2), note that if infinitely many transitions precede t_i , then some C_j must contain infinitely many transitions that precede t_i , and that these therefore must precede a transition in C_j which corresponds to the n th communication event of q for some n . This is impossible, since each transition in C_j is preceded by finitely many other transitions in C_j .

□

Proof of lemma 3.4.7.

Statement of the lemma: Let the models M_1 and M_2 be as in definition 3.4.4. Let a, b, c , and d be I/O-systems in Γ for which $h_{M_1}(a) = h_{M_1}(b)$ and $h_{M_1}(c) = h_{M_1}(d)$. Assume that

- (i) there is a trace $t \in (I(a) \cup O(a))^*$ such that $t \notin T(a)$ but $t \in T(b)$.
- (ii) $T(c) \subseteq T(d)$.

Then there exists a context Φ expressible with composition and abstraction of systems in Γ , such that

$$h_{M_2}(\Phi[a]) = h_{M_2}(c)$$

$$h_{M_2}(\Phi[b]) = h_{M_2}(d)$$

Proof. Let $E(a) = I(a) \cup O(a)$ and $E(c) = I(c) \cup O(c)$. To begin with, we shall assume that the sets $E(a)$ and $E(c)$ are disjoint. In the proof, we construct an

I/O-system N , such that the conclusion of the lemma is satisfied by the context $\Phi[\cdot] = (\cdot \| N) \setminus E(a)$.

The idea of the construction is the following. The system N communicates both via the set of events $E(a)$ of a , and the events $E(c)$ of c . The system N records the sequence that has occurred on $E(a)$, and uses this information to decide which events shall be performed on $E(c)$. The system N can always perform the sequences of events in $E(c)$ that are in $T(c)$. If the sequence of events on $E(a)$ becomes t , then N also becomes able to perform sequences of events in $E(c)$ that are in $T(d)$. Thus, when the events $E(a)$ are abstracted, it is possible to perform the sequences in $T(d)$ if and only if t can be performed as a sequence of $E(a)$.

More precisely, the network N is defined as the tuple $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$, where

$$I = O(a) \cup I(c)$$

$$O = I(a) \cup O(c)$$

Σ consists of pairs of the form $\langle t_1, t_2 \rangle$, where

$$t_1 \in (E(a))^*.$$

$$t_2 \in T(d).$$

Intuitively, the first component t_1 of the state records the sequence of events in $E(a)$ that have occurred, and the second component t_2 records the sequence of events in $E(c)$ that have occurred.

$$\sigma^0 = \langle \rangle, \langle \rangle$$

R contains all transition that are of the types below. We use e_c to range over $E(c)$, and e_a to range over $E(a)$.

- 1) $\langle t_1, t_2 \rangle \xrightarrow{e_a} \langle t_1 e_a, t_2 \rangle$ for all $t_1 \in (E(a))^*$ and $t_2 \in T(c)$. If t is a prefix of t_1 , then t_2 can also be in $T(d)$.
- 2) $\langle t_1, t_2 \rangle \xrightarrow{e_c} \langle t_1, t_2 e_c \rangle$ for all $t_2 e_c \in T(c)$.
- 3) $\langle t_1, t_2 \rangle \xrightarrow{e_c} \langle t_1, t_2 e_c \rangle$ if t is a prefix of t_1 and $t_2 e_c \in T(d)$.

Intuitively, the transitions (1) record the sequence of events that has occurred on $E(a)$. Transitions (2) state that a sequence in $T(c)$ may always occur on $E(c)$. Transitions (3) state that if the sequence t has occurred on $E(a)$, then also the sequences in $T(d)$ may occur on $E(c)$.

$\mathcal{F} = \{F\}$, where F contains all output transitions in R .

We must verify that N is indeed an I/O-system. The requirement (1) of definition 3.2.1 is satisfied, since the sets of sequences $(E(a))^*$, $T(c)$ and $T(d)$ can always be extended by input events. Requirements (2) and (3) follow directly from the definition of $\mathcal{F}$.

The conclusion in the statement of the lemma now follows from the obser-

vation that N can always perform sequences in $T(c)$ on $E(c)$. The sequences in $T(d)$ can also be performed if and only if t can be performed on $E(a)$.

In the case where the sets $E(a)$ and $E(c)$ are not pairwise disjoint, we find two new sets I, O of communication events that are disjoint from the sets $E(a)$ and $E(c)$, and two new I/O-systems f, g with input events I , output events O , and different sets of traces $T(f) \subset T(g)$. The construction of the proof can then be repeated twice, first with a, b and f, g , then with f, g and c, d .

□

Proof of Lemma 3.4.8.

Statement of lemma: Let the models $\mathcal{M}_2$ and $\mathcal{M}_3$ be as in definition 3.4.4. Let a, b, c , and d be I/O-systems in Γ for which $h_{\mathcal{M}_2}(a) = h_{\mathcal{M}_2}(b)$ and $h_{\mathcal{M}_2}(c) = h_{\mathcal{M}_2}(d)$. Assume that

- (i) there is a finite or infinite sequence $q \in (I(a) \cup O(a))^\dagger$ such that $q \notin Q(a)$ but $q \in Q(b)$.
- (ii) $Q(c) \subseteq Q(d)$.

Then there exists a context Φ expressible with composition and abstraction of systems in Γ , such that

$$h_{\mathcal{M}_3}(\Phi[a]) = h_{\mathcal{M}_3}(c)$$

$$h_{\mathcal{M}_3}(\Phi[b]) = h_{\mathcal{M}_3}(d)$$

Proof. To begin with, we shall assume that the sets $I(a)$, $O(a)$, $I(c)$, and $O(c)$ are (pairwise) disjoint. Let $E(a) = I(a) \cup O(a)$ and $E(c) = I(c) \cup O(c)$. In the proof, we construct an I/O-system N , such that the conclusion of the lemma is satisfied by the context $\Phi[\cdot] = (\cdot || N) \setminus E(a)$.

The idea of the construction is the following. The system N communicates both via the set of events $E(a)$ of a , and the events $E(c)$ of c . The system N knows the behavior of both c and d , and can behave both like c and like d . The system N records the sequence that has occurred on $E(a)$, and uses this information to decide whether to behave like c or to behave like d . The intention is that N shall behave like d if and only if the sequence of events on $E(a)$ during the completed computation is q .

However, during the computation N can never know whether q will be the sequence of events on $E(a)$ during the completed computation. Therefore N must in the beginning behave like d , and continue to behave like d as long as the sequence of events on $E(a)$ is a prefix of q . Thereafter N should switch to c if the sequence of events will not become q . This can occur if

- (i) The sequence of events on $E(a)$ is not a prefix of q , or

- (ii) The sequence of events on $E(a)$ is a proper prefix of q which is never extended.

To cover case (i), N shall switch to c if an event occurs which makes the sequence on $E(a)$ no longer a prefix of q . To cover case (ii), we introduce a fairness requirement which states that N must switch to c if the sequence on $E(a)$ stays a proper prefix of q indefinitely. A problem arises if q is infinite, and in this case we must require that N switches to c if the sequence of events on $E(a)$ remains the same indefinitely.

The I/O-system N is defined as the tuple $\langle I, O, \Sigma, \sigma^0, R, \mathcal{F} \rangle$, where

$$I = O(a) \cup I(c)$$

$$O = I(a) \cup O(c)$$

Σ consists of tuples of the form $\langle t_1, t_2, \sigma_1, \sigma_2, n \rangle$, where

t_1 is a sequence in $(E(a))^*$.

t_2 is a sequence in $T(c) = T(d)$.

σ_1 is a state of c ,

σ_2 is a state of d or the distinguished symbol $\perp$,

$n \in \{1, 2\}$.

Intuitively, the first component t_1 of the state records the sequence of events in $E(a)$ that have occurred, and the second component t_2 records the sequence of events in $E(c)$ that have occurred. The third component σ_1 simulates the state of c , and the fourth component σ_2 simulates the state of d . If $\sigma_2 \neq \perp$, then it is still possible that the quiescent trace of the computation will be q . In this case N behaves like d , but also keeps track of possible states of c in order to be able to switch to c if necessary. If $\sigma_2 = \perp$, then N has decided that q will not be the quiescent trace of the computation, and behaves only like c . Thus the component σ_2 is unimportant. The fifth component n is needed for technical reasons if q is infinite.

$\sigma^0 = \langle \langle \rangle, \langle \rangle, \sigma_1^0, \sigma_2^0, 1 \rangle$, where σ_1^0 is the initial state of c , and σ_2^0 is the initial state of d .

R contains transitions of the following types. We use e_a to range over $E(a)$, and e_c to range over $E(c) \cup \{\tau\}$.

- 1) $\langle t_1, t_2, \sigma_1, \sigma_2, n \rangle \xrightarrow{e_a} \langle t_1 e_a, t_2, \sigma_1, \sigma_2, 1 \rangle$ for all $e_a \in E(a)$, and $n \in \{1, 2\}$,
and
 $\langle t_1, t_2, \sigma_1, \sigma_2, n \rangle \xrightarrow{\tau} \langle t_1, t_2, \sigma_1, \sigma_2, n \rangle$ for all $n \in \{1, 2\}$.
- 2) $\langle t_1, t_2, \sigma_1, \sigma_2, n \rangle \xrightarrow{e_c} \langle t_1, t_2 e_c, \sigma_1', \sigma_2', n \rangle$ for $n = 1$ or 2 , if t_1 is a prefix of q , $\sigma_2 \xrightarrow{e_c} \sigma_2'$ is a transition of d , and σ_1' may occur after a sequence of transitions of c with the sequence of communication events $t_2 e_c$.

- 3) $\langle t_1, t_2, \sigma_1, \sigma_2, n \rangle \xrightarrow{\tau} \langle t_1, t_2, \sigma_1, \perp, n \rangle$ for all $t_1 \neq q$.
- 4) $\langle t_1, t_2, \sigma_1, \sigma_2, 1 \rangle \xrightarrow{\tau} \langle t_1, t_2, \sigma_1, \sigma_2, 2 \rangle$
- 5) $\langle t_1, t_2, \sigma_1, \perp, n \rangle \xrightarrow{e_c} \langle t_1, t_2 e_c, \sigma'_1, \perp, n \rangle$ if $\sigma_1 \xrightarrow{e_c} \sigma'_1$ is a transition of c .

Intuitively, transitions (1) record the events in $E(a)$. Transitions (2) correspond to the case that N behaves like d . For such transitions, also the third component of the state must be changed to record a state of c to which it is possible to switch later. Transitions (3) model the decision to behave only like c . Such a decision can occur at any moment, if the current sequence of events on $E(a)$ is not q . We intend to put the transitions 3) into a fairness set, so that a transition of type 3) must be taken if the sequence t_1 eventually becomes a stable proper prefix of q . In order that 3) need not be taken if q is infinite, and t_1 is continuously extended, we make it possible to insert transitions 4) once for each prefix of q , which are also in the same fairness set. Transitions (5) model the case where N behaves only like c .

$\mathcal{F} = \{F_1, F_2\} \cup \mathcal{F}_1 \cup \mathcal{F}_2$, where

F_1 consists of all output transitions and silent transitions in R .

F_2 consists of all transitions of form 3) and 4).

$\mathcal{F}_1$ consists of sets of transitions of form 5), one set corresponding to each fairness set of c

$\mathcal{F}_2$ consists of sets of transitions of form 2), one set corresponding to each fairness set of d .

We must verify that N is indeed an I/O-system in Γ . Requirement (1) in definition 3.2.1 follows from the fact that an input event can always be added to $(E(a))^*$ and that c and d are I/O-systems. The requirements on $\mathcal{F}$ follow directly from the definition of $\mathcal{F}$.

The conclusion of the lemma follows from the definition of N . Consider a sequence $p \in Q(c)$. A quiescent computation of N with the sequence of events p on $E(c)$ can be constructed by first performing a transition of form (4), and thereafter using transitions of form (5) and (1). In this case the sequence of events on $E(a)$ is irrelevant.

Next consider a sequence $p \in Q(d)$. We construct a quiescent computation of N in which the sequence of events on $E(a)$ is q and the sequence of events on $E(c)$ is p . There is a quiescent computation of d with the sequence of events p . The computation of N contains the corresponding sequence of transitions of form (2) as a subsequence. Transitions of form (1) are used to perform q on $E(a)$. If q is infinite, we must take care to perform infinitely many events of form (4), otherwise the fairness requirement represented by F_2 would force a transition of type (3) to occur.

Next consider a quiescent computation of N . If a transition of form 4) is performed, the sequence of events on $E(c)$ will be a quiescent trace of c , since N behaves as c after the transition of form (4). If no transition of form 4) is performed, the sequence of events on $E(c)$ will be a quiescent trace of d . This can only happen if the sequence of events on $E(a)$ is q .

The conclusion is that the sequence of events on $E(c)$ in a quiescent computation can be in $Q(d)$ iff q is the sequence of events on $E(a)$.

The case in which the sets $I(a)$, $O(a)$, $I(c)$, and $O(c)$ are not pairwise disjoint is treated in the same way as in the preceding proof of lemma 3.4.7.

□

Proof of Lemma 3.5.3

Statement of Lemma: Assume that there are two Kahn-networks $a, b \in \Upsilon$ such that $h_{M_1}(a) = h_{M_1}(b)$, and a sequence of events q such that $q \notin Q(a)$ and $q \in Q(b)$. Then there is a context Φ expressible by the composition and abstraction operations, such that

$$h_{M_H}(\Phi[a]) \neq h_{M_H}(\Phi[b])$$

Proof. The construction is similar to the proof of the preceding lemma. In the proof, we construct a Kahn-network N , such that the conclusion of the lemma is satisfied by the context $\Phi[\cdot] = (\cdot || N) \setminus (I(a) \cup O(a))$.

Let c be a channel which is not a channel of a or b , over which the message 1 can be transmitted. The input events of N are $I(N) = O(a)$ and the output events are $O(N) = I(a) \cup \{c(1)\}$. For each input channel of a there is an output channel of N , and for each output channel of a there is an input channel of N . Thus each channel of a is matched by a channel of N . In addition N contains the channel c . This situation is shown in Fig. 3.6.1 for the case where a has one input channel and one output channel.

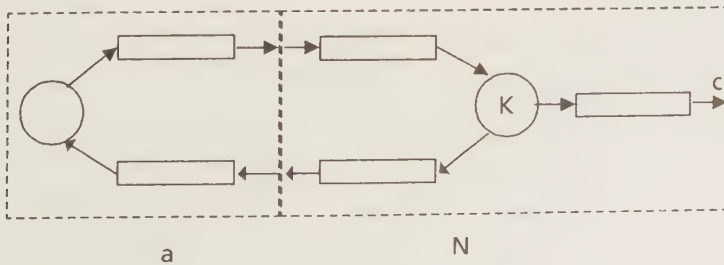


Fig. 3.6.1. Channels of a and of N .

For each channel name of a , we observe that the two channels, one of a and one of N , are connected in series, and hence equivalent to one single channel.

This follows from the requirement in the definition of Kahn-networks, that each output channel must eventually deliver each message that it contains. If the component K in Fig. 3.6.1 can always read any message from its input channels, and there is a fairness requirement which states that it will read all messages in each input channel, then the quiescent traces of a in Fig. 3.6.1. are the same as the quiescent traces of A in Fig. 3.6.2.

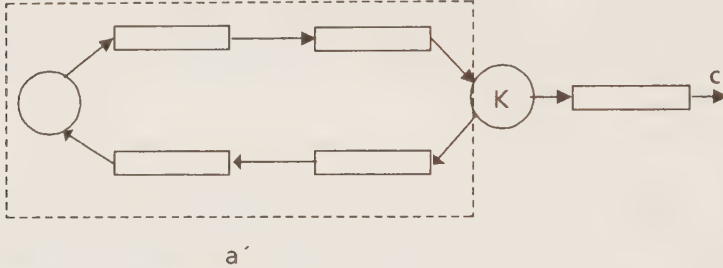


Fig. 3.6.2. Another view of a and N .

We now use lemmas 3.4.7, and 3.4.8. If a and b have different sets of traces, we use lemma 3.4.7 to construct an I/O-system K which produces different traces, hence different histories, on c , depending on whether a or b is used as the left component in Fig. 3.6.2. If a and b have the same set of traces, they must have different sets of quiescent traces. We then use lemma 3.4.8 to construct an I/O-system K which produces different quiescent traces, hence different histories, on c , depending on whether a or b is used as the left component in Fig. 3.6.2.

The requirement that K is always able to read input messages is satisfied, since K is an I/O-system. We add fairness sets to N to satisfy the requirement that K eventually reads each message in each input channel. $\square$

4 Specification and Verification

4.1 Introduction

In chapter 3, we presented a model of I/O-systems, which describes an I/O system by the set of its communication events and the set of its quiescent traces. In this chapter, we shall use the model to develop a method for specification and verification of I/O-systems.

By presenting a model of I/O-systems, the problem of specifying a system is reduced to the problem of specifying its description in the model. Our model suggests that a system be specified by

- (1) its input events and output events, and
- (2) properties of its quiescent traces.

In this chapter, we present a method for specifying a system by properties of its quiescent traces. We first present proof rules for verifying that a system conforms to a specification, given that its components satisfy their specifications. These proof rules are related to other traced-based proof systems, e.g., by Misra and Chandy ([MC 81]), by Chen and Hoare ([CH 81]), and by Nguyen et al ([NDGO 86]).

Thereafter, we present a formalism for expressing properties of quiescent traces by transition systems with liveness requirements. The transition system generates traces by sequences of transitions that satisfy the liveness requirements. An example of a transition system used to specify properties of traces is given in chapter 1.

Similar methods, in which specifications have the form of transition systems, have been presented by Lamport ([La 83]) and Stark ([St 84], [St 86]). Lamport uses transition systems to constrain the possible transitions of a system, while Stark uses transition systems to specify sequences of communication events performed by a system.

When verifying a system from specifications of components, the main idea of our method is to establish a simulation between the transition systems of the components and that of the system. To establish liveness requirements of a specification, we present proof rules that employ induction over well-founded sets. The proof rules are adapted from work for proving termination and other liveness properties by Grumberg et al ([GFMR 81]), by Lehmann, Pnueli, and Stavi ([LPS 81]), by Owicki and Lamport ([OL 82]), and by Manna and Pnueli ([MP 84]).

The proof method of Stark is similar to ours, and uses simulations between transition systems. In Lamport's method, safety properties are verified by a mapping from the states of the component specifications to the state of the system specification.

Both Lamport and Stark formulate liveness requirements in linear temporal logic. In our transition system formalism, liveness requirements state that some transition must eventually occur in a sequence of transitions that satisfies some specified condition.

An important class of I/O-systems communicate by FIFO channels or FIFO buffers. Specifications of such networks often contain state variables that represent sequences. We present techniques by which the buffers can be represented implicitly in specifications, and techniques for simplifying proofs that involve systems that communicate by FIFO channels.

In the next section, we present an abstract framework for specifying I/O-systems by properties of their quiescent traces. In section 4.3, we present a formalism for expressing properties of sequences by transition systems. In section 4.4, we present operations on transition systems that correspond to the proof rules in section 4.2. In section 4.5, we discuss completeness of the proof rules in section 4.4. For systems that communicate via FIFO buffers, we present special techniques for specification and verification in section 4.6. Section 4.7 contains proofs of some of the theorems in chapter 4.

4.2 Specification and Verification: a General Framework

In this section, we consider the general problem of specifying and verifying I/O-systems by properties of their quiescent traces. Given a formalism for expressing properties of sequences, we present a method for specification of I/O-systems and proof rules that correspond to the operations on I/O-systems defined in chapter 3.

4.2.1 Trace specifications

The model presented in section 3.3 suggests that a system be specified by

- (1) its input events and output events, and
- (2) properties of its quiescent traces.

Thus a specification consists of two parts. Part (1) can be handled, given a notation for expressing sets of communication events. Rules for composition and abstraction can be deduced directly from the rules in theorem 3.3.2. In the following, we assume that this has been taken care of, and concentrate on part (2).

The main application of our verification method is to prove that a system conforms to a specification, given that its components satisfy their specifications. The proof rules of our verification method become simpler if a specification of N can then also be used as a specification of the larger system in which N is a component. This is attained by requiring that properties of quiescent traces be formulated as properties of restrictions to (a subset of) the communication events of the specified network. For example, a specification of N which states *“each quiescent trace only contains events on the channel c ”* may be valid for N , but is in general not valid for a larger network in which N is a component, since the larger network may perform events in which N does not participate. On the other hand, if the specification is changed to *“the restriction of each quiescent trace onto the channels of N only contains events on the channel c ”*, then the specification is also valid for a larger network.

Definition 4.2.1. A trace specification S is a tuple $\langle E(S), Q(S) \rangle$, where

$E(S)$ is a set of communication events,

$Q(S)$ is a subset of $E(S)^\dagger$.

□

Intuitively, the trace specification $S = \langle E(S), Q(S) \rangle$ is intended to specify a property of sequences. This property is satisfied by a sequence q for which $q[E(S) \in Q(S)]$. Using this interpretation, we can define logical operations on trace specifications.

Definition 4.2.2. Let $S_1, \dots, S_k$ be trace specifications.

- The *conjunction* $\bigwedge_{i=1}^k S_i$ of $S_1, \dots, S_k$ is the trace specification S defined by $S = \langle E(S), Q(S) \rangle$ where

$$E(S) = \bigcup_{i=1}^k E(S_i)$$

$$Q(S) = \{q \in (E(S))^+ \mid q[E(S_i) \in Q(S_i) \text{ for all } i]\}$$

- The *implication* $S_1 \supset S_2$ between the trace specifications S_1 and S_2 holds iff

$$E(S_2) \subseteq E(S_1) \text{ and}$$

$$Q(S_1)[E(S_2) \subseteq Q(S_2).$$

□

Notation. For an I/O-system N , we shall in the following sometimes use the notation $E(N)$ for $I(N) \cup O(N)$.

Definition 4.2.3. A network N denoted by $[N] = I(N), O(N), Q(N)$ satisfies a trace specification $S = \langle E(S), Q(S) \rangle$, written $N \text{ sat } S$ if

- 1) $E(S) \subseteq I(N) \cup O(N)$.
- 2) for each $q \in Q(N)$, it holds that $q[E(S) \in Q(S)$.

□

Intuitively, the formula $N \text{ sat } S$ states that *the restriction of each quiescent trace of N to the set of events $E(S)$ is in the set $Q(S)$* . A specification of a system N should also give the sets of input events and output events of N . When writing a formula $N \text{ sat } S$, we assume that these sets are implicitly understood. Rules for obtaining input events and output events of the composition or abstraction of systems can be derived from theorem 3.3.2.

4.2.2 Proof rules

Using the operations on trace specifications defined in the preceding subsection, we formulate the following proof rules for proving formulas of the form $N \text{ sat } S$. The proof rules are similar to the proof rules in our earlier work ([Jo 85]). The proof rules are also related to rules in other trace-based verification methods, e.g., by Misra and Chandy ([MC 81]), by Chen and Hoare ([CH 81]), and by Nguyen et al ([NDGO 86]).

Proof rules for $N \text{ sat } S$

Composition: Let $N_1, \dots, N_k$ be compatible I/O-systems and $S_1, \dots, S_k$ be trace specifications

$$\frac{N_i \text{ sat } S_i \quad i = 1, \dots, k}{N_1 \parallel \dots \parallel N_k \text{ sat } \bigwedge_{i=1}^k S_i}$$

Abstraction: Let E be a set of output events of N which is disjoint from $E(S)$.

$$\frac{N \text{ sat } S}{N \setminus E \text{ sat } S}$$

Consequence:

$$\frac{N \text{ sat } S \quad S \supset S'}{N \text{ sat } S'}$$

Soundness of the proof rules follows from the following theorem.

Theorem 4.2.4. The preceding proof rules for $N \text{ sat } S$ are sound.

Proof. The proof uses the rules for composition and abstraction in theorem 3.3.2. We treat each rule separately.

Composition: The requirements on $N_1 \parallel \dots \parallel N_k \text{ sat } \bigwedge_{i=1}^k S_i$ in definition 4.2.3 are motivated as follows:

- 1) By the premises of the rule, $E(S_i) \subseteq E(N_i)$ for each i (Note that we use $E(N_i)$ for $(I(N_i) \cup O(N_i))$). By the definition of conjunction of trace specifications $E(\bigwedge_{i=1}^k S_i) = \bigcup_{i=1}^k E(S_i)$. Since $E(N_1 \parallel \dots \parallel N_k) = \bigcup_{i=1}^k E(N_i)$ we conclude $E(\bigwedge_{i=1}^k S_i) \subseteq E(N_1 \parallel \dots \parallel N_k)$,
- 2) Let q be a quiescent trace of $N_1 \parallel \dots \parallel N_k$. By theorem 3.3.2, $q \upharpoonright (E(N_i))$ is a quiescent trace of N_i for each i . By the premises of the rule we conclude that $(q \upharpoonright (E(N_i))) \upharpoonright E(S_i) \in Q(S_i)$ for each i . But $(q \upharpoonright (E(N_i))) \upharpoonright E(S_i) = q \upharpoonright E(S_i)$, since $E(S_i) \subseteq E(N_i)$. Hence $q \upharpoonright E(S_i) \in Q(S_i)$ for each i , whence by the definition of conjunction of trace specifications we have $q \upharpoonright E(S) \in Q(\bigwedge_{i=1}^k S_i)$.

Abstraction: The requirements on $N \setminus E \text{ sat } S$ are motivated as follows:

- 1) Immediate, since E is disjoint from $E(S)$.
- 2) Let q be a quiescent trace of $N \setminus E$. By theorem 3.3.2, there is a quiescent trace $q' \in Q(N)$ such that $q = q' \setminus E$. The premise of the rule states that $q' \upharpoonright E(S) \in Q(S)$. Since $E \cap E(S) = \emptyset$, it follows that $q' \upharpoonright E(S) = q \upharpoonright E(S)$, whence $q \upharpoonright E(S) \in Q(S)$.

Consequence: Immediate.

□

Example 4.2.5. To illustrate the use of the proof rules, we present a simple example of how properties of a system are proved from properties of its components. We shall consider two systems N_1 and N_2 , shown in Fig. 4.2.1. System N_1 has the input channel α , and the output channel β , system N_2 has the input channel β and the output channel γ . Messages from a set M are sent over the channels. Both N_1 and N_2 act like buffers, i.e., they copy the messages that arrive on the input channel to the output channel in the same order.

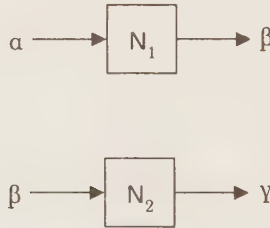


Fig. 4.2.1. Two systems, N_1 and N_2 .

We shall show that if we connect N_1 , and N_2 , and abstract the internal channel β , the result will be a system as shown in Fig. 4.2.2, which also behaves like a buffer.

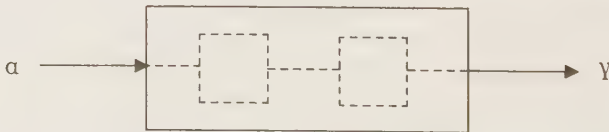


Fig. 4.2.2. The result of connecting N_1 and N_2 to form $(N_1 \parallel N_2) \setminus \{M_\beta\}$.

A communication event $\alpha(m)$ denotes the transmission of message m over channel α . Let us denote by M_α the set of communication events $\{\alpha(m) \mid m \in M\}$ and similarly for M_β and M_γ . We use the notation $msgs(q, \alpha)$ to denote

the sequence of messages in events of the form $\alpha(m)$ in the sequence q . For the verification, we use the specifications

$$N_1 \text{ sat } \langle M_\alpha \cup M_\beta, \{q \mid \text{msg}(q, \alpha) = \text{msg}(q, \beta)\} \rangle$$

$$N_2 \text{ sat } \langle M_\beta \cup M_\gamma, \{q \mid \text{msg}(q, \beta) = \text{msg}(q, \gamma)\} \rangle$$

The specification of N_1 states that the sequence of messages transmitted over α is equal to the sequence of messages transmitted over β . We intend to show that

$$(N_1 \parallel N_2) \backslash M_\beta \text{ sat } \langle M_\alpha \cup M_\gamma, \{q \mid \text{msg}(q, \alpha) = \text{msg}(q, \gamma)\} \rangle$$

We first use the composition rule to obtain

$$N_1 \parallel N_2 \text{ sat } \langle M_\alpha \cup M_\beta \cup M_\gamma, \{q \mid \left[\begin{array}{l} \text{msg}(q, \alpha) = \text{msg}(q, \beta) \wedge \\ \text{msg}(q, \beta) = \text{msg}(q, \gamma) \end{array} \right] \} \rangle$$

The consequence rule gives us

$$N_1 \parallel N_2 \text{ sat } \langle M_\alpha \cup M_\gamma, \{q \mid \text{msg}(q, \alpha) = \text{msg}(q, \gamma)\} \rangle$$

Finally, the abstraction rule gives us

$$(N_1 \parallel N_2) \backslash M_\beta \text{ sat } \langle M_\alpha \cup M_\gamma, \{q \mid \text{msg}(q, \alpha) = \text{msg}(q, \gamma)\} \rangle$$

since $M_\beta \cap (M_\alpha \cup M_\gamma) = \emptyset$. $\square$

4.2.3 Completeness issues

In this subsection, we investigate the following question: Assume that $N \text{ sat } S$, that N is obtained from the networks $N_1, \dots, N_k$ by the composition and abstraction operations, and that $N_i \text{ sat } S_i$ is true for $i = 1, \dots, k$. Is it then possible to deduce $N \text{ sat } S$ by the proof rules for $N \text{ sat } S$?

If the trace specifications $S_1, \dots, S_k$ of $N_1, \dots, N_k$ are too weak, we may not be able to prove $N \text{ sat } S$. To answer the above question in the affirmative, we must therefore require that the specifications $S_1, \dots, S_i$ of $N_1, \dots, N_k$ are as strong as possible.

Definition 4.2.6. Let N be an I/O-system with $\llbracket N \rrbracket = \langle I(N), O(N), Q(N) \rangle$. Let S be a trace specification $\langle E(S), Q(S) \rangle$. The formula $N \text{ sat } S$ is *precise* if

- 1) $E(S) = I(N) \cup O(N)$, i.e., the set of communication events of N is the set of communication events of S .

- 2) $Q(N) = Q(S)$, i.e., the specification S allows exactly all the quiescent traces of N .

□

Theorem 4.2.7. Assume that the network N is obtained from the networks $N_1, \dots, N_k$ by application of the composition and abstraction operations, and that for each N_i the trace specification S_i is such that the formula $N_i \text{ sat } S_i$ is precise. If $N \text{ sat } S$ holds, then this formula can be proven from the formulas $N_i \text{ sat } S_i$ and the proof rules for $N \text{ sat } S$.

Proof. We shall prove that the proof rules for composition and abstraction preserve preciseness, i.e., if the formulas in the premises are precise, then a precise formula can be obtained as conclusion. By induction on the construction of N , it then follows that we can prove a precise formula $N \text{ sat } S'$ for N . This means that $\langle E(S'), Q(S') \rangle = \langle I(N) \cup O(N), Q(N) \rangle$. We then use the definition of $N \text{ sat } S$ to conclude $E(S) \subseteq E(S')$, and $Q(S')[E(S) \subseteq Q(S)]$, whence $S' \supset S$ is true, and $N \text{ sat } S$ can be obtained by the consequence rule.

The proof that the rules for composition and abstraction preserve preciseness goes as follows

Composition: Assume that the formulas $N_i \text{ sat } S_i$ are precise for $i = 1, \dots, k$. We claim that $N_1 \parallel \dots \parallel N_k \text{ sat } \bigwedge_{i=1}^k S_i$ is precise. We must verify the conditions in definition 4.2.5.

1) Immediate.

- 2) Since $Q(\bigwedge_{i=1}^k S_i) = \{q \in (E(S))^\dagger \mid \text{for all } i \ q[E(S_i) \in Q(S_i)]\}$, it follows by theorem 3.3.2 that $Q(N_1 \parallel \dots \parallel N_k) = Q(\bigwedge_{i=1}^k S_i)$.

Abstraction: Assume that the formula $N \text{ sat } S$ is precise, and that E is subset of $O(N)$. Define the trace specification $S \setminus E = \langle E(S \setminus E), Q(S \setminus E) \rangle$ by $E(S \setminus E) = E(S) - E$ and $Q(S \setminus E) = \{q \setminus E \mid q \in Q(S)\}$. From the preciseness of $N \text{ sat } S$, it is easy to verify that the formula $N \setminus E \text{ sat } S \setminus E$ is precise. We can now use the proof rules to deduce $N \setminus E \text{ sat } S \setminus E$ by the following two steps:

- (1) Use the consequence rule to prove $N \text{ sat } S \setminus E$. It is not difficult to verify that the necessary premise $S \supset S \setminus E$ holds.
- (2) Use the abstraction rule to prove $N \setminus E \text{ sat } S \setminus E$. The necessary premise $N \text{ sat } S \setminus E$ has been established in step (1), and it is obvious that E is disjoint from $E(S \setminus E)$.

□

The preceding theorem states that $N \text{ sat } S$ can be proven from formulas $N_i \text{ sat } S_i$ that are precise. However, this is not longer true if the assumption of preciseness is dropped. To illustrate this, we shall present an example of two trace specifications S and S' , such that whenever $N \text{ sat } S$ is true, it also holds that $N \text{ sat } S'$, but for which $S \supset S'$ does not hold.

Example 4.2.8. Let i be an input event, and o be an output event. Consider the trace specifications $S = \langle \{i, o\}, Q(S) \rangle$ and $S' = \langle \{i, o\}, Q(S') \rangle$ where $Q(S)$ is the set of sequences q of i and o that satisfy

$$q[\{i\}] \neq \langle \rangle \supset q[\{o\}] = \langle \rangle$$

and $Q(S')$ is the set of sequences q of i and o that satisfy

$$q[\{o\}] = \langle \rangle.$$

Assume that a system N satisfies $N \text{ sat } S$. We claim that N also satisfies $N \text{ sat } S'$. This follows from the fact that if $N \text{ sat } S$ is true, then N never produces any output. Namely, if N produced an event o then, since N can always accept input, an input event i may occur, and the resulting computation would violate the trace specification S . Thus, the formula $N \text{ sat } S'$ is also true. However, the implication $S \supset S'$ is not true.

The problem of incompleteness of trace-based proof systems has been studied by Widom, Gries, and Schneider ([WGS 87]) for safety-properties. They suggest to add axioms that characterize properties that hold for the set of traces of any system. Nguyen et al ([NDGO 86]) assume that the proof system contains a set of primitive systems, for each of which there is a specification which is as strong as possible.

4.3 Trace Generators

In this section, we present a formalism for expressing trace specifications. A trace specification is expressed by a *trace generator*. Intuitively, a trace generator is a transition system with liveness requirements, which generates traces in sequences of transitions that satisfy the liveness requirements. We shall assume that communication events consists of an event name and a parameter. For instance, in example 4.2.5 we would consider an event to consist of an event name α and a parameter m .

When defining a trace generator, we assume a set $\mathcal{E}$ of *event names*, which does not include the silent event τ .

We assume a first-order language with predicate, function, and constant symbols, and equality $=_D$.

We assume a set of variables. The set of variables is partitioned into the following subsets:

$V^{event} = \{e \mid e \in \mathcal{E}\}$ is a set of propositional variables ranging over truth-values.

Intuitively, the variable e is true if the event e occurs.

$V^{par} = \{v_e \mid e \in \mathcal{E}\}$ is a set of variables that are intended to range over parameters of events, where v_e ranges over the parameter of the event e .

V^{state} is a set of state variables

$V^{trans} = \{x^{old} \mid x \in V^{state}\} \cup \{x^{new} \mid x \in V^{state}\}$ are used to express properties of transitions.

V is a set of auxiliary variables.

Terms and formulas are built from variables, predicate, constant and function symbols by the usual first-order connectives. For a term or formula t , let $FV(t)$ denote the set of free variables in t .

An *interpretation* is a mapping from the set of variables to their appropriate domains. Interpretations are extended to terms and formulas in the usual manner.

A *state formula* ϕ is a formula with $FV(\phi) \subseteq V^{state} \cup V$. A *transition formula* ϕ is a formula with $FV(\phi) \subseteq V^{event} \cup V^{par} \cup V^{trans} \cup V$

Definition 4.3.1. A *trace generator* is a tuple $\langle E, \bar{x}, \Theta, \mathcal{T}, \mathcal{F} \rangle$ where

$E \subseteq \mathcal{E}$

$\bar{x}$ is a tuple of *state variables* $x_1, \dots, x_n$ in V^{state} .

Θ is an *initial predicate*, which is a state formula.

$\mathcal{T}$ is a finite set of *transition types*. A transition type in $\mathcal{T}$ is of the form

$$e(v_e) \quad ; \quad c \longrightarrow \bar{x} := f$$

where

$e \in E \cup \{\tau\}$.

c is a formula with $FV(c) \subseteq V^{state} \cup \{v_e\} \cup V$

f is a term with $FV(f) \subseteq V^{state} \cup \{v_e\} \cup V$

If e is the distinguished event τ , then there is no parameter variable v_e in $e(v_e)$, c or f .

$\mathcal{F}$ is a set of *liveness requirements*. Each liveness requirement is either

- a *guarantee requirement* $\phi \hookrightarrow \psi$ where ϕ is a state formula and ψ is a transition formula
- a *fairness requirement* $\langle \phi, \psi \rangle$ where ϕ is a state formula or a transitions formula and ψ is a state formula or a transition formula.

□

Intuitively, a guarantee requirement $\phi \hookrightarrow \psi$ states that if ϕ holds in a sequence of transitions, then ψ must hold at some future point. A fairness requirement $\langle \phi, \psi \rangle$ states that if ϕ holds infinitely often, then ψ holds infinitely often.

We will often use the notation

$$e(t) \quad ; \quad c \longrightarrow \bar{x} := f$$

where t is a term with $v_e \notin FV(t)$ as a shorter way to denote the transition type

$$e(v_e) \quad ; \quad c \wedge (v_e =_D t) \longrightarrow \bar{x} := f$$

Definition 4.3.2. Let $\mathcal{A}$ be the trace generator $\langle E, \bar{x}, \Theta, \tau, \mathcal{F} \rangle$.

- A *state* of $\mathcal{A}$ is a tuple $\bar{\xi} = \langle \xi_1, \dots, \xi_n \rangle$, where each ξ_i is taken from the domain of interpretation of the variable x_i .
- A *communication event* of $\mathcal{A}$ is a pair $e(d)$, where $e \in E$, and d belongs to the domain of interpretation of v_e .
- A *transition* of $\mathcal{A}$ is a labeled transition $\bar{\xi} \xrightarrow{e(d)} \bar{\xi}'$ where $\bar{\xi}$ and $\bar{\xi}'$ are states of $\mathcal{A}$, $e(d)$ is τ or a communication event of $\mathcal{A}$, and such that either
 - (i) there is a transition type $e(v_e) \quad ; \quad c \longrightarrow \bar{x} := f$ in $\mathcal{A}$ and an interpretation I such that

$$I(\bar{x}) = \bar{\xi} \quad I(c) \text{ is true} \quad I(v_e) = d \quad I(f) = \bar{\xi}'$$

- (ii) $\bar{\xi} =_D \bar{\xi}'$ and $e(d)$ is τ . Such a transition is referred to as a *stuttering transition*.

- Let ϕ be a state formula. The state $\langle \xi_1, \dots, \xi_n \rangle$ is a ϕ -state if $I(\phi)$ is true for all interpretations I such that $I(x_i) = \xi_i$ for $i = 1, \dots, n$.
- Let ψ be a transition formula. The transition $\bar{\xi} \xrightarrow{e(d)} \bar{\xi}'$ is a ψ -transition if $I(\psi)$ is true for all interpretations I such that

$$I(x_i^{old}) = \xi_i \text{ and } I(x_i^{new}) = \xi'_i \text{ for } i = 1, \dots, n$$

$$I(v_e) = d$$

$$I(e) \text{ is true, and } I(e') \text{ is false for all other event names } e' \in E.$$

- let C be a sequence of transitions of $\mathcal{A}$. A state (transition) formula ϕ holds *infinitely often* in C if C contains infinitely many ϕ -states (ϕ -transitions), or if C is finite and ends in a ϕ -state.

Definition 4.3.3. A *computation* C of the trace generator $\mathcal{A}$ is a (finite or infinite) sequence of transitions

$$\bar{\xi}^0 \xrightarrow{e^1} \bar{\xi}^1 \xrightarrow{e^2} \dots \xrightarrow{e^n} \bar{\xi}^n \xrightarrow{e^{n+1}} \dots$$

(for brevity we omit the parameters from events) which fulfills the following requirements:

- (A) *Initialization:* $\bar{\xi}^0$ is a Θ -state.
- (B) *State to state sequencing:* Each transition $\bar{\xi}^i \xrightarrow{e^{i+1}} \bar{\xi}^{i+1}$ is a transition of $\mathcal{A}$.
- (C) *Guarantee:* For each guarantee requirement $\phi \hookrightarrow \psi$ of $\mathcal{A}$, each ϕ -state is followed by a ψ -transition somewhere later in C .
- (D) *Fairness:* For each fairness requirement $\langle \phi, \psi \rangle$ of $\mathcal{A}$, if ϕ holds infinitely often in C , then ψ holds infinitely often in C .

A *partial computation* of $\mathcal{A}$ is a sequence of transitions which fulfills conditions (A) and (B). For a sequence of transitions C , let $\Xi(C)$ denote the sequence of communication events of $\mathcal{A}$ that label transitions in C .

□

The meaning of a trace generator is given by the following definition.

Definition 4.3.4. Let $\mathcal{A}$ be a trace generator. The meaning $\llbracket \mathcal{A} \rrbracket$ of $\mathcal{A}$ is the trace specification defined by $\llbracket \mathcal{A} \rrbracket = \langle E(\llbracket \mathcal{A} \rrbracket), Q(\llbracket \mathcal{A} \rrbracket) \rangle$ such that

$E(\llbracket \mathcal{A} \rrbracket)$ is the set of communication events of $\mathcal{A}$

$Q(\llbracket \mathcal{A} \rrbracket)$ is the set $\{\Xi(C) \mid C \text{ is a computation of } \mathcal{A}\}$ of sequences of communication events in computations of $\mathcal{A}$.

□

Example 4.3.5. As an example, we present a trace generator that specifies a FIFO buffer. Suppose that the buffer has input events on the channel α , and output events on the channel β . The communication events of the buffer are of form $\alpha(m)$ or $\beta(m)$ for some parameter m , taken from some set of values. We present a trace specification which states that the sequence of events of form $\beta(m)$ is the same as the sequence of events of form $\alpha(m)$, and that each event on β occurs after its corresponding event on α . The state of the trace specification is a sequence of values that have occurred on α , but not yet on β . We use the following notation for predicates and functions on finite sequences:

$\langle \rangle$ denotes the empty sequence.

$s \bullet m$ denotes the result of adding the element m to the end of the sequence s .

$tl(s)$ denotes the result of deleting the first element of the sequence s .

$hd(s)$ is the first element of the sequence s .

The buffer can then be specified by the following trace generator. When displaying the set of event names, we shall also include their parameter(s) for readability. In this case, we use v for the parameter of events.

Events:	$\alpha(v), \beta(v)$	
Variables:	x	
Initialization:	$x = \langle \rangle$	
Transitions:	$\alpha(v) : true \longrightarrow x := x \bullet v$	(T1)
	$\beta(hd(x)) : x \neq \langle \rangle \longrightarrow x := tl(x)$	(T2)
Liveness:	$x \neq \langle \rangle \hookrightarrow \beta$	

The liveness requirement states that if a message has been received on α but not delivered on β , then eventually an event must occur on channel β .

□

4.4 Reasoning with Trace Generators

In this section, we present verification techniques for the case when trace generators are used to specify properties of quiescent traces. In the proof system of section 4.2, the operations $\wedge$ (conjunction), and $\supset$ (implication) are used in verifications. In this section, we therefore present corresponding constructions on trace generators.

4.4.1 Product of trace generators

In this subsection, we present an operation, product, on trace generators which corresponds to the conjunction operation on trace specifications. Recall that if $S_1, \dots, S_k$ are behavior specifications, then their conjunction is defined as $S = \langle E(S), Q(S) \rangle$, where

$$E(S) = \bigcup_{i=1}^k E(S_i)$$

$$Q(S) = \{q \in (E(S))^\dagger \mid q \upharpoonright E(S_i) \in Q(S_i) \text{ for all } i\}.$$

Definition 4.4.1. (Product of Trace Generators) Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be a set of trace generators, where $\mathcal{A}_i = \langle E_i, \bar{x}_i, \Theta_i, \tau_i, \mathcal{F}_i \rangle$. We assume that no variable, except those in V^{event} and V^{par} , occurs in two different $\mathcal{A}_i$'s. Their *product* $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_k)$ is the trace specification $\mathcal{A} = \langle E, \bar{x}, \Theta, \tau, \mathcal{F} \rangle$, where

$$E = E_1 \cup \dots \cup E_k.$$

$$\bar{x} = \bar{x}_1, \dots, \bar{x}_k.$$

$$\Theta = \Theta_1 \wedge \dots \wedge \Theta_k.$$

τ is obtained from τ_i for $i = 1, \dots, k$ by synchronizing transitions with a common communication event. For an event name $e \in E$, let $\mathcal{A}_{i_1}, \dots, \mathcal{A}_{i_m}$ be the trace specifications for which $e \in E_{i_j}$ for $j = 1, \dots, m$. Let

$$e(v_e) \quad : \quad c_{i_j} \longrightarrow \bar{x}_{i_j} := f_{i_j}$$

be a transition type in τ_{i_j} for $j = 1, \dots, m$. Then there is a transition type in τ of form

$$e(v_e) \quad : \quad c_{i_1} \wedge \dots \wedge c_{i_m} \longrightarrow \left\{ \begin{array}{c} \bar{x}_{i_1} := f_{i_1} \\ \vdots \\ \bar{x}_{i_m} := f_{i_m} \end{array} \right\}$$

Transitions with the silent event τ are not synchronized; thus each silent transition in each τ_i is also a transition in τ .

$$\mathcal{F} = \mathcal{F}_1 \cup \dots \cup \mathcal{F}_k.$$

□

Intuitively, the product operation is similar to the composition of transition systems. In a computation of $\mathcal{A}$, the trace generators $\mathcal{A}_1, \dots, \mathcal{A}_k$ execute in parallel and synchronize on common communication events. Thus the part of the computation which corresponds to $\mathcal{A}_i$ checks that each sequence $q \in Q(\llbracket \mathcal{A} \rrbracket)$ satisfies $q \models E(\llbracket \mathcal{A}_i \rrbracket) \in Q(\llbracket \mathcal{A}_i \rrbracket)$.

The following theorem states that product of trace generators corresponds to conjunction of trace specifications.

Theorem 4.4.2. Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be a set of trace generators. It holds that

$$\llbracket \Pi(\mathcal{A}_1, \dots, \mathcal{A}_k) \rrbracket = \bigwedge_{i=1}^k \llbracket \mathcal{A}_i \rrbracket$$

Proof. The proof is somewhat similar to the proof of theorem 3.3.2. The details are given in section 4.7 □

4.4.2 Proving Implication between trace generators.

In this subsection, we present a rule for establishing an implication of form $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$ when $\mathcal{A}_i$ is the trace generator $\langle E_i, \bar{x}_i, \Theta_i, \tau_i, \mathcal{F}_i \rangle$ for $i = 1, 2$. Recall that when S_1 and S_2 are behavior specifications, then the implication $S_1 \supset S_2$ is defined by

$$E(S_2) \subseteq E(S_1), \text{ and}$$

$$Q(S_1) \upharpoonright E(S_2) \subseteq Q(S_2).$$

The first condition is established by proving that $E_2 \subseteq E_1$. To establish the second condition, it must be proven that for each computation C_1 of $\mathcal{A}_1$ there is a computation C_2 of $\mathcal{A}_2$ such that $\Xi(C_2) = \Xi(C_1) \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)$.

The idea of the method is to prove that for each computation C_1 of $\mathcal{A}_1$

$$\bar{\xi}_1^0 \xrightarrow{e^1} \bar{\xi}_1^1 \xrightarrow{e^2} \dots \xrightarrow{e^n} \bar{\xi}_1^n \xrightarrow{e^{n+1}} \dots$$

there is a computation C_2 of $\mathcal{A}_2$

$$\bar{\xi}_2^0 \xrightarrow{e^1} \bar{\xi}_2^1 \xrightarrow{e^2} \dots \xrightarrow{e^n} \bar{\xi}_2^n \xrightarrow{e^{n+1}} \dots,$$

which simulates C_1 step-by-step. This means that the n th transition of C_1 is related to the n th transition of C_2 in such a way that $\epsilon^n = e^n \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)$, i.e., $\epsilon^n = e^n$ if e^n is a communication event of $\mathcal{A}_2$, otherwise $\epsilon^n = \tau$.

The rule reduces a proof about sequences of transitions to a proof which reasons about individual transitions. The price for this simplification is that the rule is incomplete in the general case. In section 4.5, we investigate under which conditions the rule is complete.

Definition 4.4.3. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be trace generators for which $E_2 \subseteq E_1$. A *simulation relation* R is a relation between the states of $\mathcal{A}_1$ and the states of $\mathcal{A}_2$ such that

- (1) for each initial state $\bar{\xi}_1^0$ of $\mathcal{A}_1$, there is an initial state $\bar{\xi}_2^0$ of $\mathcal{A}_2$ such that $R(\bar{\xi}_1^0, \bar{\xi}_2^0)$.
- (2) Whenever $R(\bar{\xi}_1, \bar{\xi}_2)$ holds and $\bar{\xi}_1 \xrightarrow{e^n} \bar{\xi}_1'$ is a transition of $\mathcal{A}_1$, there exists a state $\bar{\xi}_2'$ of $\mathcal{A}_2$ such that

$$\bar{\xi}_2 \xrightarrow{e^n \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)} \bar{\xi}_2' \text{ is a transition of } \mathcal{A}_2, \text{ and } R(\bar{\xi}_1', \bar{\xi}_2').$$

□

Given a simulation relation R between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$, we can by induction over n conclude that for each computation C_1 of $\mathcal{A}_1$ there exists a partial computation C_2 of $\mathcal{A}_2$ which simulates C_1 step-by-step in the manner described. To complete the proof of the implication, we must verify that C_2

is indeed a computation, i.e., that the liveness requirements of C_2 are satisfied. To express the requirements involved, it is helpful to consider the transition sequences C_1 and C_2 together.

Definition 4.4.4. Let R be a simulation relation between the states of the trace generators A_1 and A_2 .

- a *joint R -state* is a pair $\langle \bar{\xi}_1, \bar{\xi}_2 \rangle$, where $\bar{\xi}_1$ and $\bar{\xi}_2$ are states of A_1 and A_2 such that $R(\bar{\xi}_1, \bar{\xi}_2)$ holds.
- A *joint R -transition* is a transition $\langle \bar{\xi}_1, \bar{\xi}_2 \rangle \xrightarrow{e(d)} \langle \bar{\xi}'_1, \bar{\xi}'_2 \rangle$ between joint R -states such that $\bar{\xi}_1 \xrightarrow{e(d)} \bar{\xi}'_1$ is a transition of A_1 and $\bar{\xi}_2 \xrightarrow{e(d)[E(\|A_2\|)]} \bar{\xi}'_2$ is a transition of A_2

□

Definition 4.4.4 can be regarded as the construction of a transition system, in which the states are joint R -states, and the transitions are joint R -transitions. We can extend the definitions of ϕ -states and ψ -transitions to joint R -states and joint R -transitions. For instance, a joint R -state is a ϕ -state if $I(\phi)$ is true for all interpretations I such that $I(\bar{x}_1) = \bar{\xi}_1$ and $I(\bar{x}_2) = \bar{\xi}_2$. A ϕ -transition is defined analogously. For sequences of joint R -transitions, we can define liveness requirements in the same way as for trace generators. In particular, a liveness requirement in A_1 or A_2 can also be used as a liveness requirement for sequences of joint R -transitions.

Definition 4.4.5. Let R be a simulation relation between the states of the trace generators A_1 and A_2 . Let P and U be state formulas, and T a transition formula.

- The formula

$$(P) \xrightarrow[R]{} \circ (T ; U)$$

denotes that for each joint R -transition $\langle \bar{\xi}_1, \bar{\xi}_2 \rangle \xrightarrow{e(d)} \langle \bar{\xi}'_1, \bar{\xi}'_2 \rangle$ such that the joint R -state $\langle \bar{\xi}_1, \bar{\xi}_2 \rangle$ is a P -state, either $\langle \bar{\xi}_1, \bar{\xi}_2 \rangle \xrightarrow{e(d)} \langle \bar{\xi}'_1, \bar{\xi}'_2 \rangle$ is a T -transition, or $\langle \bar{\xi}'_1, \bar{\xi}'_2 \rangle$ is a U -state.

- Let $\mathcal{F}$ be a set of liveness requirements. The formula

$$(P) \xrightarrow[R]{\mathcal{F}} \diamond (T ; U)$$

denotes that each (finite or infinite) sequence of joint R -transitions

$$\langle \bar{\xi}_1^0, \bar{\xi}_2^0 \rangle \xrightarrow{e^1} \langle \bar{\xi}_1^1, \bar{\xi}_2^1 \rangle \xrightarrow{e^2} \dots \xrightarrow{e^n} \langle \bar{\xi}_1^n, \bar{\xi}_2^n \rangle \xrightarrow{e^{n+1}} \dots$$

which satisfies all liveness requirements in $\mathcal{F}$ and for which $\langle \bar{\xi}_1^0, \bar{\xi}_2^0 \rangle$ is a P -state, it is the case that the sequence contains either a T -transition or a U -state.

□

A formula of the form $(P) \xrightarrow[R]{\circ} (T ; U)$ can be proven by checking all joint R -transitions. In subsection 4.4.3, we give rules for proving formulas of the form $(P) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; U)$.

Theorem 4.4.6. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two trace generators. The implication $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$ holds if

- (1) $E_2 \subseteq E_1$, and
- (2) There exists a simulation R between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$ such that
 - (i) For each guarantee requirement $\phi \hookrightarrow \psi \in \mathcal{F}_2$,

$$(\phi) \xrightarrow[R]{\mathcal{F}_1} \Diamond (\psi ; \text{false})$$

- (ii) For each fairness requirement $\langle \phi, \psi \rangle \in \mathcal{F}_2$

$$(\text{true}) \xrightarrow[R]{\mathcal{F}'_1} \Diamond (\text{false} ; \psi) \text{ if } \psi \text{ is a state formula}$$

$$(\text{true}) \xrightarrow[R]{\mathcal{F}'_1} \Diamond (\psi ; \text{false}) \text{ if } \psi \text{ is a transition formula}$$

where $\mathcal{F}'_1 = \mathcal{F}_1 \cup \{\langle \text{true}, \phi \rangle\}$.

Proof. By induction on the length of C_1 we conclude that for each computation C_1 of $\mathcal{A}_1$ with the sequence of events q there is a corresponding partial computation C_2 of $\mathcal{A}_2$ with the sequence of events $q[E(\llbracket \mathcal{A}_2 \rrbracket)]$. We must verify that C_2 is a computation if C_1 is a computation.

By induction on the length of C_1 we can also conclude that there is a sequence C of joint R -transitions, such that C_1 is the sequence of first components of C and C_2 is the sequence of second components of C . If C_1 is a computation, then C satisfies the liveness requirements $\mathcal{F}_1$ of $\mathcal{A}_1$. We verify that C satisfies the liveness requirements of $\mathcal{A}_2$ as follows:

- (i) Assume that $\phi \hookrightarrow \psi \in \mathcal{F}_2$ is a guarantee requirement of $\mathcal{A}_2$. From condition (2)(i) we conclude that each ϕ -state in C is followed by a ψ -transition. Since ϕ and ψ only refer to the sequence of second components of C , the same holds for C_2 ,
- (ii) Assume that $\langle \phi, \psi \rangle \in \mathcal{F}_2$ is a fairness requirement of $\mathcal{A}_2$. From condition (2)(ii) we conclude that if ϕ holds infinitely often in C then there is always a later point at which ψ holds. Since this holds for sequences of joint R -transitions that start in any joint R -state in C , we conclude that ψ is true

infinitely often in C . By the same reasoning as in case (i), we conclude that C_2 satisfies the fairness requirement $\langle \phi, \psi \rangle$

□

4.4.3 Proof rules for auxiliary formulas.

In this section, we present a set of rules for establishing formulas of the form $(P) \xrightarrow[\mathcal{R}]{\mathcal{F}} \Diamond (T ; U)$, where T is a transition formula and U is a state formula. The rules are based on the work by Manna and Pnueli ([MP 84]) and by Owicki and Lamport ([OL 82]) for proving liveness properties of concurrent programs, and the work for proving termination of programs under fairness conditions by Grumberg, Francez, Makowsky, and de Roeper ([GFMR 81]), and by Lehman, Pnueli, and Stavi ([LPS 81]).

We first recall that a well-founded relation $<$ on a set W is a binary relation for which there is no infinite strictly decreasing sequence

$$w_1 > w_2 > w_3 > \dots$$

of elements in W .

Proof rules for formulas of the form $(P) \xrightarrow[\mathcal{R}]{\mathcal{F}} \Diamond (T ; U)$

Guarantee: Assume that $\phi \hookrightarrow \psi \in \mathcal{F}$, where ϕ is a state formula, and ψ is a transition formula.

$$\begin{array}{c} (P) \xrightarrow[\mathcal{R}]{} \circ (T ; P \vee U) \\ (P) \xrightarrow[\mathcal{R}]{} \circ (\psi \supset T ; U) \\ P \supset (\phi \vee U) \\ \hline (P) \xrightarrow[\mathcal{R}]{\mathcal{F}} \Diamond (T ; U) \end{array}$$

Fairness: Assume that $\langle \phi, \psi \rangle \in \mathcal{F}$. We give the proof rule in the case where ϕ and ψ are both transition formulas. The other cases are analogous. Let $\mathcal{F}' = \mathcal{F} \setminus \{\langle \phi, \psi \rangle\}$.

$$\begin{array}{c} (P) \xrightarrow[\mathcal{R}]{} \circ (T ; P \vee U) \\ (P) \xrightarrow[\mathcal{R}]{} \circ (\psi \supset T ; U) \\ (P) \xrightarrow[\mathcal{R}]{\mathcal{F}'} \Diamond (\phi \vee T ; U) \\ \hline (P) \xrightarrow[\mathcal{R}]{\mathcal{F}} \Diamond (T ; U) \end{array}$$

Chain: Let $\langle W, < \rangle$ be a well-founded structure, let $\chi(w)$ be a state formula parametrized by an element $w \in W$.

$$\frac{(\chi(w)) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; (\exists w' < w) \chi(w') \vee U) \quad \text{all } w \in W}{((\exists w) \chi(w)) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; U)}$$

Guarantee II: Assume that $\phi \hookrightarrow \psi \in \mathcal{F}$, where ϕ is a state formula, and ψ is a transition formula. Then

$$(\phi) \xrightarrow[R]{\mathcal{F}} \Diamond (\psi ; U)$$

Implication:

$$\frac{P \supset U}{(P) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; U)}$$

Consequence:

$$\frac{(P' \wedge R) \supset P \quad (P) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; U) \quad T \supset T' \quad (U \wedge R) \supset U'}{(P') \xrightarrow[R]{\mathcal{F}} \Diamond (T' ; U')}$$

Theorem 4.4.7. The preceding proof rules for formulas of the form

$$(P) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; U)$$

are sound.

Proof. We treat each rule separately.

Guarantee: Let C be a sequence of joint R -transitions which starts in a P -state. We must prove that C contains a U -state or a T -transition.

The proof is by contradiction. Assume that C does not contain any U -state or T -transition. The proof has three steps:

- (i) From the first premise, it follows that all states in C are P -states.
- (ii) By the third premise, it follows that the initial state of C is a ϕ -state. Hence by the guarantee requirement $\phi \hookrightarrow \psi$ it follows that C contains a ψ -transition.
- (iii) By (i), all states in C are P -states. The second premise now implies that the ψ -transition in C is either also a T -transition, or leads to a U -state. Hence we have derived a contradiction.

Fairness: The proof is the same as the proof of the guarantee rule, except that step (ii) is replaced by

- (ii) Since C satisfies the liveness requirements of $\mathcal{F}$, it must also satisfy the liveness requirements in $\mathcal{F}'$. By the third premise, it follows that each P -state is followed by a ϕ -transition, for the transition formula ϕ . Hence there are infinitely many transitions in C that satisfy ϕ . By the fairness requirement $\langle \phi, \psi \rangle$ we conclude that C contains ψ -transition.

Chain: Let C be a sequence of joint R -transitions, in which the first joint R -state is a $(\exists w)\chi(w)$ -state. To obtain a contradiction, assume that C contains no U -state or T -transition. From the premise of the rule, we see that for each $w \in W$, each $\chi(w)$ -state is followed by a $\chi(w')$ -state for some $w' \prec w$. It follows that there is an infinite decreasing sequence $w_1 \succ w_2 \succ w_3 \succ \dots$ of elements in W such that C contains a $\chi(w_i)$ -state. This contradicts the fact that the relation $\prec$ is well-founded, and we have a contradiction.

Guarantee II: Obvious.

Implication: Obvious.

Consequence: Obvious.

□

Example 4.4.8. As an illustration, we shall prove that the parallel composition of two unbounded FIFO buffers behave like one unbounded FIFO buffer. A trace generator that specifies a FIFO buffer with input channel α and output channel β has been given in example 4.4.6. We shall now show that if we compose that FIFO buffer with a buffer with input channel β and output channel γ , we get a FIFO buffer with input channel α and output channel γ .

We then specify the second buffer by an analogous trace generator, and obtain their product as the trace generator Π below.

Events:	$\alpha(v), \beta(v), \gamma(v)$	
Variables:	x_1, x_2	
Initialization:	$x_1 = x_2 = \langle \rangle$	
Transitions:	$\alpha(v) : \text{true} \longrightarrow x_1 := x_1 \bullet v$	(T1)
	$\beta(hd(x_1)) : x_1 \neq \langle \rangle \longrightarrow \{x_1 := tl(x_1), x_2 := x_2 \bullet hd(x_1)\}$	(T2)
	$\gamma(hd(x_2)) : x_2 \neq \langle \rangle \longrightarrow x_2 := tl(x_2)$	(T3)
Liveness:	$x_1 \neq \langle \rangle \hookrightarrow \beta$	
	$x_2 \neq \langle \rangle \hookrightarrow \gamma$	

We shall prove that this product implies the trace generator $\mathcal{A}_{buf}$ below:

Events:	$\alpha(v), \gamma(v)$	
Variables:	y	
Initialization:	$y = \langle \rangle$	
Transitions:	$\alpha(v) : \text{true} \longrightarrow y := y \bullet v$	(V1)
	$\gamma(hd(y)) : y \neq \langle \rangle \longrightarrow y := tl(y)$	(V2)
Liveness:	$y \neq \langle \rangle \hookrightarrow \gamma$	

We must find a simulation relation R which satisfies the requirements in theorem 4.3.5. The relation R is given by $y = x_2 \bullet x_1$. We check that R is really a simulation.

- (1) For the initial states we get the condition $\langle \rangle = \langle \rangle \bullet \langle \rangle$ which is true.
- (2) To verify that R is preserved by each transition of the product, we see that the transition (T1) is simulated by (V1), (T2) by a stuttering transition, and (T3) is simulated by (V2). The conditions to be verified are:

$$[y = x_2 \bullet x_1] \supset [y \bullet v = x_2 \bullet (x_1 \bullet v)] \quad (T1)$$

$$[(y = x_2 \bullet x_1) \wedge (x_1 \neq \langle \rangle)] \supset [y = ((x_2 \bullet hd(x_1)) \bullet tl(x_1))] \quad (T2)$$

$$[(y = x_2 \bullet x_1) \wedge (x_2 \neq \langle \rangle)] \supset [tl(y) = tl(x_2) \bullet x_1] \quad (T3)$$

which are all simple properties of finite sequences.

We next establish the liveness requirements of the trace generator. In this case there is one guarantee requirement, namely $y \neq \langle \rangle \hookrightarrow \gamma$. According to theorem 4.3.5, we must establish the formula

$$(y \neq \langle \rangle) \xrightarrow[\mathcal{R}]{\mathcal{F}_1} \Diamond (\gamma ; false)$$

by the proof rules for establishing liveness properties. The proof uses the chain rule. We shall illustrate such a proof by a *proof diagram*, similar to the diagram proofs by Manna and Pnueli ([MP 84]), and the proof lattices by Owicki and Lamport ([OL 82]). For each value of the parameter w , the proof diagram contains a node labeled by the state formula $\chi(w)$. For a premise of the rule of the form $(\chi(w)) \xrightarrow[\mathcal{R}]{\mathcal{F}} \Diamond (T ; (\exists w' \prec w) \chi(w') \vee U)$ the diagram contains arrows from the node corresponding to $\chi(w)$ to the nodes $\chi(w')$, where w' is a possible candidate for the lower value of the parameter w' . If the premise is established by an application of the guarantee or the fairness rule, the arrow is labeled by the corresponding liveness requirement in $\mathcal{F}$. The proof diagram for $(y \neq \langle \rangle) \xrightarrow[\mathcal{R}]{\mathcal{F}_1} \Diamond (\gamma ; false)$ then becomes:

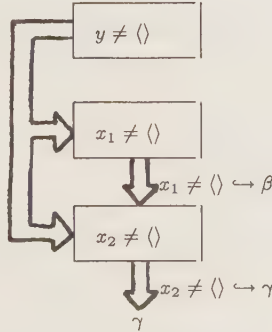


Fig. 4.4.1. Proof diagram for liveness property of example 4.4.8..

The premise corresponding to the arrows from the formula $y \neq \langle \rangle$ follows by the implication rule. The premises that correspond to the two other arrows follow by the guarantee rule applied to the proper guarantee requirement.

Remark. As the reader notices, the above verification of the composition of two buffers is tedious. We would like to do less work in order to verify a simple example as this one. In section 4.6, techniques will be presented that facilitate verifications of systems whose specifications involve buffer variables.

□

4.5 Completeness Issues

In this section, we discuss completeness issues for the rule in theorem 4.4.6 for proving implication between trace specifications given by trace generators, and for the rules for proving formulas of the form $(P) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; U)$. We shall here only deal with what is known as *semantical completeness* (see for instance Francez [Fr 86]). For the implication rule in theorem 4.4.6, we shall prove that a relation R exists, such that the premises of theorem 4.4.6 hold. We are only interested in R as a set of pairs of the form $\langle \xi_1, \xi_2 \rangle$, and ignore the question whether R can be expressed in any language for expressing state formulas and transition formulas.

In his work on verification and specification by transition systems, Stark ([St 84], [St 86]) establishes a theorem related to our theorem 4.5.3. Our result in theorem 4.5.4 is inspired by the work on semantic completeness of proof rules for termination under fairness conditions by Grumberg et al ([GFMR 81]), also reported by Francez ([Fr 86]).

Definition 4.5.1. A trace generator $\mathcal{A}$ is *deterministic* if it has only one initial state, no silent transition types, and at most one transition of form $\bar{\xi} \xrightarrow{e(d)} \bar{\xi}'$ for each state $\bar{\xi}$ and communication event $e(d)$.

□

Definition 4.5.2. A trace generator $\mathcal{A}$ is *deadlock-free* if

- (1) It has at least one initial state.
- (2) Each partial computation of $\mathcal{A}$ can be extended to a computation of $\mathcal{A}$.

□

Theorem 4.5.3. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two trace generators. If $\llbracket \mathcal{A}_1 \rrbracket \supseteq \llbracket \mathcal{A}_2 \rrbracket$, and

- (1) $\mathcal{A}_1$ is deadlock-free,
- (2) $\mathcal{A}_2$ is deterministic, and
- (3) $(\phi_i) \xrightarrow{R} \circ (\psi_i ; \phi_i)$, for each guarantee requirement $\phi_i \hookrightarrow \psi_i$ of $\mathcal{A}_1$.

then there exists a simulation relation R between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$, in the set-theoretical sense, such that the premises of theorem 4.4.6 hold.

Proof. Given in section 4.7.

□

Remark. Note that we do not say anything about whether the relation R can be expressed in our language.

Theorem 4.5.4. Let R be a simulation relation between the states of the two trace specifications $\mathcal{A}_1$ and $\mathcal{A}_2$. Assume that

- (1) $(\phi_i) \xrightarrow{R} \circ (\psi_i ; \phi_i)$, for each guarantee requirement $\phi_i \hookrightarrow \psi_i$ of $\mathcal{A}_1$.
- (2) $\mathcal{A}_1$ and $\mathcal{A}_2$ have only finitely many liveness requirements

Then the rules for auxiliary formulas on pages 72-73 are semantically complete.

Proof. The proof is adapted from Francez ([Fr 86]). It is given in section 4.7. □

We will now present three simple examples that show why the requirements in theorem 4.5.3 are necessary. We first give an example in which $\mathcal{A}_1$ is not deadlock-free, thereafter an example in which $\mathcal{A}_2$ is not deterministic, and then an example in which $(\phi_i) \xrightarrow{R} \circ (\psi_i ; \phi_i)$ does not hold for some guarantee requirement $\phi_i \hookrightarrow \psi_i$.

Example 4.5.5. To motivate condition (1) of theorem 4.5.3, consider two trace specifications $\mathcal{A}_1$ and $\mathcal{A}_2$, both of which have the event name a . The parameter of a is irrelevant. The states and transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ are displayed in Fig. 4.5.1, with an arrow pointing to the initial state.

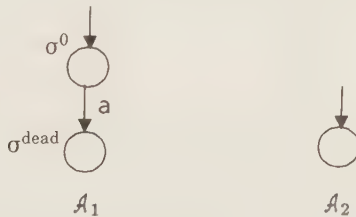


Fig. 4.5.1. States and transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$.

$\mathcal{A}_1$ has one guarantee-requirement: $x = \sigma^{dead} \hookrightarrow x = \sigma^0$, whereas $\mathcal{A}_2$ has no liveness requirements. There is only one computation of $\mathcal{A}_1$, namely the computation with only the initial state and no transitions. If the transition were taken, then it would be impossible to satisfy the guarantee requirement. Therefore $Q(\llbracket \mathcal{A}_1 \rrbracket)$ is the set $\{\langle \rangle\}$ consisting only of the empty sequence. The same holds for $Q(\llbracket \mathcal{A}_2 \rrbracket)$. Thus $\llbracket \mathcal{A}_1 \rrbracket$ is equivalent to $\llbracket \mathcal{A}_2 \rrbracket$. However, there is no simulation between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$, since $\mathcal{A}_2$ contains no transition labeled by a . $\square$

Example 4.5.6. To motivate condition (2) of theorem 4.5.3, consider the two trace generators $\mathcal{A}_1$ and $\mathcal{A}_2$, both with event names a , b , and c , and states and transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ are shown in Fig. 4.5.2.

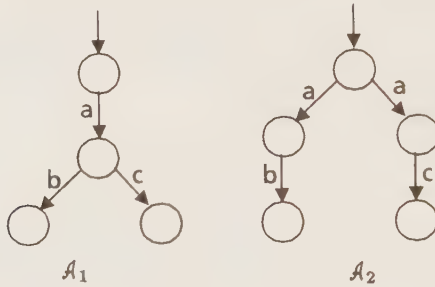


Fig. 4.5.2. States and transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$.

It is obvious that $\llbracket \mathcal{A}_1 \rrbracket \equiv \llbracket \mathcal{A}_2 \rrbracket$, since both define the same set of sequences. However, there is no simulation between the states of $\mathcal{A}_1$ and the states of $\mathcal{A}_2$. The reason is that the “second” state of $\mathcal{A}_1$ cannot be simulated by any single state of $\mathcal{A}_2$. To verify that $\mathcal{A}_1 \supset \mathcal{A}_2$, we would have to simulate that state by both of the two middle states of $\mathcal{A}_2$. $\square$

Example 4.5.7. To motivate condition (3), consider $\mathcal{A}_1$ and $\mathcal{A}_2$, both with event names a and b , both with the state-transition diagram of Fig. 4.5.3.

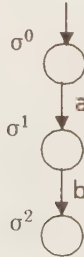


Fig. 4.5.3. States and transitions of both $\mathcal{A}_1$ and $\mathcal{A}_2$.

$\mathcal{A}_1$ has the guarantee requirement $\sigma^1 \hookrightarrow b$, whereas $\mathcal{A}_2$ has the guarantee requirement $\sigma^2 \hookrightarrow b$. Clearly, $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$, since $Q(\llbracket \mathcal{A}_1 \rrbracket) = \{\langle a, b \rangle\}$ and $Q(\llbracket \mathcal{A}_2 \rrbracket) = \{\langle \rangle, \langle a, b \rangle\}$. There is an obvious simulation between the states of $\mathcal{A}_1$ and the states of $\mathcal{A}_2$. However, to prove the liveness requirement of $\mathcal{A}_2$, we must according to theorem 4.5.3 establish

$$(\text{current state is } \sigma^2) \xrightarrow[R]{\mathcal{F}_1} \Diamond (b ; \text{false})$$

but this formula does not hold. The reason is that there is a sequence of joint R -states which violates this formulas, namely the sequence which consists of the only joint R -state $\langle \sigma^2, \sigma^2 \rangle$. This sequence satisfies the guarantee requirement of $\mathcal{A}_1$, since the state σ^1 never occurs. $\square$

4.6 Specifying communication over FIFO channels.

An important class of distributed systems communicate by passing messages over unbounded FIFO channels. When specifying properties of such systems by trace generators, state variables that denote sequences are used to model the states of the FIFO channels. This use of state variables was illustrated in example 4.4.8. We observed that the verification was somewhat tedious. In this section, we present techniques for simplifying specification and verification of systems that communicate via unbounded FIFO channels.

In his work on decidability problems for finite-state machines that communicate over FIFO channels, Pacht ([Pac 82]) gives conditions under which certain sequences of transitions can be ignored when analyzing reachability or deadlock properties. This work is related to our theorem 4.6.12.

In subsection 4.6.1, we present an extension of trace generators for specifying systems that communicate via unbounded FIFO channels. In subsection 4.6.2, we present techniques for simplifying the verification of systems that communicate via unbounded FIFO channels.

4.6.1 Buffered trace generators

In this subsection, we extend the definition of trace generators to include primitives for unbounded FIFO channels. The extension is called *buffered trace generator*.

Definition 4.6.1. A *buffered trace generator* (BTG) is a tuple $\mathcal{B} = \langle E, \bar{x}, \Theta, \mathcal{T}, \mathcal{F} \rangle$, in which the components are the same as the components of a trace generator, defined in definition 4.3.1, with the following extensions:

- (1) Each event name in E may have an attribute, either *input buffered* or *output buffered*. Event names with such attributes are called *buffered*. All other event names are called *unbuffered*.

(2) A transition type may have more than one event name. We use the notation

$$e_1(v_{e_1}) / \dots / e_l(v_{e_l}) \quad ; \quad c \longrightarrow \bar{x} := f$$

to denote a transition with the event names $e_1, \dots, e_l$. In such a transition, both $FV(c)$ and $FV(f)$ are subsets of $V^{state} \cup \{v_{e_1}, \dots, v_{e_l}\}$.

(3) $\mathcal{F}$ may contain the following extra liveness requirements:

For each buffered event name a requirement of *buffer guarantee*

For each input buffered event name a requirement of *buffer fairness*.

□

Definition 4.6.2. A BTG is *safe* if each transition type has at most one unbuffered event name.

□

Intuitively, if an event name e is buffered, then the state of the BTG contains a variable which denotes a FIFO queue of parameters of e . Communication with the environment is performed via the corresponding FIFO queues. If an event name has a buffer guarantee requirement, then each element that is put into the corresponding buffer must eventually be removed from the buffer. If an event name has a buffer fairness requirement, then if infinitely often it is possible to remove an element from the buffer, then infinitely often an element is removed from the buffer.

The meaning of a safe buffered trace generator $\mathcal{B}$ is defined formally by transforming it into an equivalent trace generator. We present a transformation which *expands* a buffered event type by adding the corresponding buffer as an explicit state variable, adding the necessary transitions, etc. Thus, a safe BTG $\mathcal{B}$ can be transformed to an equivalent trace generator by expanding all its buffered event names.

Definition 4.6.3. Let $\mathcal{B}$ be a buffered trace generator $\mathcal{B} = \langle E, \bar{x}, \Theta, \mathcal{T}, \mathcal{F} \rangle$. Let the event name e be buffered in $\mathcal{B}$. Then the meaning $\llbracket \mathcal{B} \rrbracket$ of $\mathcal{B}$ is the same as the meaning $\llbracket \mathcal{B}' \rrbracket$ of the BTG $\mathcal{B}'$, defined by *expanding* the event name e in $\mathcal{B}$ to obtain $\mathcal{B}' = \langle E, \bar{x}', \Theta', \mathcal{T}', \mathcal{F}' \rangle$, where

$\bar{x}' = \bar{x}, buf_e$, where buf_e is a state variable whose intended interpretation is a sequence of elements from the domain of interpretation of the variable v_e .

$\Theta' = \Theta \wedge buf_e = \langle \rangle$.

τ' is obtained from τ . If e is input buffered, then τ' is obtained from τ as follows:

- (i) First transform all transition types in τ

$$e_1(v_{e_1}) / \dots / e(v_e) / \dots / e_l(v_{e_l}) \quad ; \quad c \longrightarrow \bar{x} := f$$

with the event name e into

$$e_1(v_{e_1}) / \dots / \dots / e_l(v_{e_l}) \quad ; \quad \left[\begin{array}{l} buf_e \neq \langle \rangle \wedge \\ c[hd(buf_e)/v_e] \end{array} \right] \longrightarrow \left\{ \begin{array}{l} \bar{x} := f[hd(buf_e)/v_e] \\ buf_e := tl(buf_e) \end{array} \right\}$$

where $c[hd(buf_e)/v_e]$ is the result of replacing all free occurrences of v_e in c by $hd(buf_e)$, and similarly for $f[hd(buf_e)/v_e]$. The transformation removes the event name e from the transition type. Instead of performing a communication event $e(v_e)$, an element is removed from the buffer buf_e . In the case where e is the only event name of the transition type, then the transformed transition type has the event τ .

- (ii) Thereafter add a transition type of the form

$$e(v_e) \quad ; \quad true \longrightarrow buf_e := buf_e \bullet v_e$$

If e is output buffered, then τ' is obtained from τ as follows.

- (i) First transform all transition types in τ

$$e_1(v_{e_1}) / \dots / e(v_e) / \dots / e_l(v_{e_l}) \quad ; \quad c \longrightarrow \bar{x} := f$$

with the event name e into

$$e_1(v_{e_1}) / \dots / \dots / e_l(v_{e_l}) \quad ; \quad c \longrightarrow \left\{ \begin{array}{l} \bar{x} := f \\ buf_e := buf_e \bullet v_e \end{array} \right\}.$$

i.e., the event name e is removed from the events of the transition type. Instead of performing a communication event $e(v_e)$, an element is added to the buffer buf_e . and the communication is instead performed with the buffer buf_e . In the case where e is the only event name of the transition type, then the transformed transition type has the event τ .

- (ii) Then add a transition type of the form

$$e(hd(buf_e)) \quad ; \quad buf_e \neq \langle \rangle \longrightarrow buf_e := tl(buf_e)$$

$\mathcal{F}'$ is obtained from $\mathcal{F}$ by the following additions

- (i) if e has a buffer guarantee requirement, the requirement

$$buf_e \neq \langle \rangle \hookrightarrow buf_e^{new} = tl(buf_e^{old})$$

is added, which states that all elements of the buffer must eventually be removed.

- (ii) if e is input buffered with a buffer fairness requirement, the requirement

$$\langle enabled(e) \wedge buf_e \neq \langle \rangle, buf_e^{new} = tl(buf_e^{old}) \rangle$$

is added, where $enabled(e)$ is the disjunction of the **enabling conditions** of the transition types in $\mathcal{T}$ that have the event name e .

□

If $\mathcal{B}$ is a safe BTG, then $\mathcal{B}'$ is also a safe BTG. Thus, the meaning of a safe BTG can be obtained by expanding all its buffered event names to obtain an (unbuffered) trace generator.

Example 4.6.4. As an illustration, we specify the buffer in example 4.3.5 by a buffered trace generator. In this case, the state variable is represented by an input buffer. We must only specify a transition which removes the first element of the buffer and produces an output event. Thus there is only one state, and we do not need any state variables. These are therefore omitted from the buffered trace generator.

Events:	$\alpha(v)$ input buffered $\beta(v)$
Transitions:	$\alpha(v) / \beta(v)$
Liveness:	α : buffer guarantee.

Fig. 4.6.1. Buffered trace generator specifying a buffer.

To obtain an equivalent unbuffered trace generator, we expand the buffered event name α . The result is shown in Fig. 4.6.2.

Events:	$\alpha(v)$ $\beta(v)$
Variables:	buf_α
Initialization:	$buf_\alpha = \langle \rangle$
Transitions:	$\alpha(v) ; true \longrightarrow buf_\alpha := buf_\alpha \bullet v$ $\beta(v) ; buf_{\alpha(v)} \neq \langle \rangle \longrightarrow buf_\alpha := tl(buf_\alpha)$
Liveness:	$buf_\alpha \neq \langle \rangle \hookrightarrow buf_\alpha^{new} := tl(buf_\alpha^{old})$

Fig. 4.6.2. Expansion of the event $\alpha(v)$ in the BTG of Fig. 4.6.1.

□

4.6.2 Reasoning about Buffered Trace Generators

In this subsection, we present techniques that can be used to simplify proofs that involve buffered trace generators. We first sketch the ideas with an example, and thereafter present the general technique.

Example 4.6.5. Consider the problem of verifying that for the buffered trace generators $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}$ we have $\llbracket \Pi(\mathcal{B}_1, \mathcal{B}_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$. We illustrate the product of the two buffered trace generators in Fig. 4.6.3, where circles denote buffered trace generators without the implicit buffers, and rectangles denote the implicit buffers.

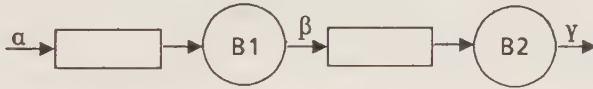


Fig. 4.6.3. Conjunction of two buffered trace generators, $\mathcal{B}_1$ and $\mathcal{B}_2$

We shall verify that the product of the BTGs in Fig. 4.6.3 implies the BTG in Fig. 4.6.4.

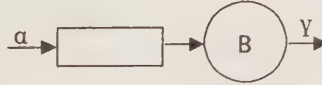


Fig. 4.6.4. Buffered trace generator $\mathcal{B}$.

To verify that $\llbracket \Pi(\mathcal{B}_1, \mathcal{B}_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$, the events on channel β are not relevant. We can therefore remove the buffer corresponding to channel β from the conjunction in Fig. 4.6.3 without affecting the truth of the implication $\llbracket \Pi(\mathcal{B}_1, \mathcal{B}_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$. Let $\mathcal{B}'_2$ denote the result of removing the buffer from $\mathcal{B}_2$. We need then only establish the implication between $\llbracket \Pi(\mathcal{B}_1, \mathcal{B}'_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$, where the structure of $\Pi(\mathcal{B}_1, \mathcal{B}'_2)$ is shown in Fig. 4.6.5.



Fig. 4.6.5. Product of two buffered trace generators, $\mathcal{B}_1$ and $\mathcal{B}'_2$.

When establishing this implication, it is unnecessary to consider the remaining buffer of $\Pi(\mathcal{B}_1, \mathcal{B}'_2)$. Namely, we remove the input buffer of $\mathcal{B}_1$, obtaining $\mathcal{B}'_1$, and prove that $\llbracket \Pi(\mathcal{B}'_1, \mathcal{B}'_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$. This implication is preserved if we add an extra input buffer both to $\Pi(\mathcal{B}'_1, \mathcal{B}'_2)$ and to $\mathcal{B}$. Adding an input buffer to $\Pi(\mathcal{B}'_1, \mathcal{B}'_2)$ yields $\Pi(\mathcal{B}_1, \mathcal{B}'_2)$. Adding an input buffer to $\mathcal{B}$ yields $\mathcal{B}$, since two buffers in parallel are equivalent to one buffer. We therefore conclude that $\llbracket \Pi(\mathcal{B}_1, \mathcal{B}'_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$

holds, and finally (since the removal of the buffer corresponding to β was harmless) conclude that $\llbracket \Pi(\mathcal{B}_1, \mathcal{B}_2) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$ holds. $\square$

This was an outline of the idea. We now proceed to the general case. We shall establish two theorems. One theorem corresponds to the removal of the buffer at β in the previous example, and the other theorem corresponds to the removal of the buffer at α .

We first consider the case that corresponds to the removal of the buffer at β in the last example. Assume a set of buffered trace generators $\mathcal{B}_1, \dots, \mathcal{B}_k$, and $\mathcal{B}$, for which we intend to prove the implication $\llbracket \Pi(\mathcal{B}_1, \dots, \mathcal{B}_k) \rrbracket \supset \llbracket \mathcal{B} \rrbracket$. The idea roughly is that a set of buffers in $\llbracket \Pi(\mathcal{B}_1, \dots, \mathcal{B}_k) \rrbracket$ can be replaced by a single buffer without affecting the truth of the implication, if

- two event names of $\mathcal{B}$ that previously were separated by at least one buffer are still separated by at least one buffer.
- We do not remove a cycle of buffers.

This rough idea will now be made precise in the following.

Definition 4.6.6. Let $\mathcal{B}_1, \dots, \mathcal{B}_k$, and $\mathcal{B}$ be buffered trace generators. Let E be the union of the event names of $\mathcal{B}_1, \dots, \mathcal{B}_k$. Assume that the event names of $\mathcal{B}$ is a subset of E . The *dependency graph* of $\mathcal{B}_1, \dots, \mathcal{B}_k$ relative to $\mathcal{B}$ is a directed graph with one node for each $\mathcal{B}_i$, one node for each event name $e \in E$, and the following edges:

- (1) If e is input buffered in $\mathcal{B}_i$, there is an edge $e \longrightarrow \mathcal{B}_i$.
- (2) If e is output buffered in $\mathcal{B}_i$, there is an edge $\mathcal{B}_i \longrightarrow e$.
- (3) If e is an event name of $\mathcal{B}_i$ which is not buffered, there are edges $e \longrightarrow \mathcal{B}_i$ and $\mathcal{B}_i \longrightarrow e$.
- (4) Between any two events names e_1 and e_2 that are event names of $\mathcal{B}$ there are edges $e_1 \longrightarrow e_2$ and $e_2 \longrightarrow e_1$.

$\square$

Intuitively, an edge $n_1 \longrightarrow n_2$ represents possible ordering constraints between events in a computation of the product of $\mathcal{B}_1, \dots, \mathcal{B}_k$ which performs a certain sequence of events of $\mathcal{B}$. The edges (1) and (2) represent the fact that events that add an element to a buffer must occur before the event that removes the element from the buffer. The two edges under (3) state that unbuffered events are synchronized with transitions of the trace generator. Under (4) we represent potential ordering constraints between events of $\mathcal{B}$ in a sequence of communication events of $\mathcal{B}$.

Definition 4.6.7. Let $\mathcal{B}$ be a BTG with the buffered event name e . The buffer buf_e is *removed* from $\mathcal{B}$ by dropping the attribute input (output) buffered for e , and dropping the requirements of buffer guarantee and fairness (if any) for e . $\square$

Definition 4.6.8. Let $\mathcal{B}_i$ be a safe BTG with the buffered event name e . The buffer buf_e is *live* in $\mathcal{B}_i$ if each element which is inserted into buf_e during a computation of $\mathcal{B}_i$ is also removed from buf_e in the same computation. $\square$

Proposition 4.6.9. Let $\mathcal{B}_i$ be a safe BTG with buffered event name e . The buffer buf_e is live in $\mathcal{B}_i$ if either of the following conditions hold

- (1) The event name e has a buffer guarantee requirement.
- (2) The event name e has a buffer fairness requirement, and $enabled(e)$ hold infinitely often in each computation of $\mathcal{B}_i$, where $enabled(e)$ is the disjunction of all enabling conditions for transitions that remove an element from buf_e .

$\square$

We shall extend the product operation to BTG's which are not necessarily safe, i.e., BTG's that are obtained by removing buffers from safe BTGs. We first need an auxiliary definition.

Definition 4.6.10. Let $\mathcal{B}_1, \dots, \mathcal{B}_k$ be BTG's. A *transition chain* of $\mathcal{B}_1, \dots, \mathcal{B}_k$ is either a silent transition type of one $\mathcal{B}_i$, or a set of transition types from $\mathcal{B}_1, \dots, \mathcal{B}_k$ such that

- (1) There is at most one transition type from each $\mathcal{B}_i$
- (2) If one transition in the transition chain has the event name e , where e is not τ , then each $\mathcal{B}_i$ with the event name e has a transition type in the transition chain.
- (3) There is no nonempty proper subset of the transition chain that satisfies conditions (1) and (2).

$\square$

We can now define the product operation for buffered trace generators.

Definition 4.6.11. Let $\mathcal{B}_1, \dots, \mathcal{B}_k$ be buffered trace generators. The product $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ is defined as the BTG obtained by performing the following two steps:

- (1) For each event name e and BTG $\mathcal{B}_i$, if e is a buffered event name in $\mathcal{B}_i$ which also occurs in some other $\mathcal{B}_j$, then expand the event name e in $\mathcal{B}_i$.

- (2) Let $B'_1, \dots, B'_k$ be the BTGs obtained after step (1). $\Pi(B_1, \dots, B_k)$ is defined as the BTG $B = \langle E, \bar{x}, \Theta, \mathcal{T}, \mathcal{F} \rangle$, where $E, \bar{x}, \Theta$, and $\mathcal{F}$ are obtained as in definition 4.4.1 of product for unbuffered trace generators. An event type with an attribute in B'_i retains the same attribute in $\Pi(B_1, \dots, B_k)$.

The transition types $\mathcal{T}$ are obtained as follows:

For each transition chain of $B'_1, \dots, B'_k$, which consists of transition types of the form

$$e_{i_1}(v_{e_{i_1}}) / \dots / e_{i_l}(v_{e_{i_l}}) \quad : \quad c_i \longrightarrow \bar{x}_i := f_i$$

for $i = 1, \dots, m$ there is a transition in $\mathcal{T}$ of form

$$e_1(v_{e_1}) / \dots / e_l(v_{e_l}) \quad : \quad c_1 \wedge \dots \wedge c_m \longrightarrow \left\{ \begin{array}{c} \bar{x}_1 := f_1 \\ \vdots \\ \bar{x}_m := f_m \end{array} \right\}$$

where $e_1, \dots, e_l$ is the union of all event names that occur in the transition chain. $\square$

Note that in the case where $B_1, \dots, B_k$ are unbuffered trace generators, this definition is equivalent to definition 4.4.1. In that case, a transition chain with event name e consists of one transition from each trace generator with e as an event name. The rule for obtaining the transition types of the product is then the same in definition 4.6.11 as in definition 4.4.1.

We now formulate the main theorem of this section: a theorem which gives the conditions under which buffers may be removed from $B_1, \dots, B_k$ without affecting the truth of the implication $\llbracket \Pi(B_1, \dots, B_k) \rrbracket \supset \llbracket B \rrbracket$.

Theorem 4.6.12. Let $B_1, \dots, B_k$ and B be safe BTGs, such that the event name f of B is a subset of the union of event names of $B_1, \dots, B_k$. Let $B'_1, \dots, B'_k$ be obtained by removing some buffers in $B_1, \dots, B_k$. Let $\mathcal{G}$ be the dependency graph of $B_1, \dots, B_k$ relative to B , and $\mathcal{G}'$ be the dependency graph of $B'_1, \dots, B'_k$ relative to B . The two implications $\llbracket \Pi(B_1, \dots, B_k) \rrbracket \supset \llbracket B \rrbracket$ and $\llbracket \Pi(B'_1, \dots, B'_k) \rrbracket \supset \llbracket B \rrbracket$ are equivalent if

- (1) for each cycle in $\mathcal{G}'$ which has more than two nodes and does not visit any node more than once, there is a corresponding cycle in $\mathcal{G}$ which visits corresponding nodes in the same order.
- (2) If buf_e is removed from B_i to obtain B'_i , then buf_e is live in B_i .

Proof. Given in section 4.7.

$\square$

In the next theorem, we formulate conditions under which buffers may be disregarded in proofs of implications.

Theorem 4.6.13. Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two safe buffered trace generators, such that the event name e is input buffered in both $\mathcal{B}_1$ and $\mathcal{B}_2$ or output buffered in both $\mathcal{B}_1$ and $\mathcal{B}_2$. Let $\mathcal{B}'_1$ denote the result of removing buf_e in $\mathcal{B}_1$. If

- (1) $\mathcal{B}'_1$ is safe
- (2) $\llbracket \mathcal{B}'_1 \rrbracket \supset \llbracket \mathcal{B}_2 \rrbracket$
- (3) if $\mathcal{B}_2$ has a buffer guarantee or buffer fairness requirement for e , then buf_e is live in $\mathcal{B}_1$

then $\llbracket \mathcal{B}_1 \rrbracket \supset \llbracket \mathcal{B}_2 \rrbracket$.

Proof. Given in section 4.7. $\square$

Example 4.6.14. We conclude this section with a somewhat more complicated example that illustrates how buffers can be removed. This example will subsequently be used in section 5.2 to facilitate the verification of the sliding window protocol in HDLC.

In the example we define five buffered trace generators, $\mathcal{B}(P_1)$, $\mathcal{B}(P_2)$, $\mathcal{B}(M_{12})$, $\mathcal{B}(M_{21})$, and $\mathcal{B}(S)$. The involved event names are X , Y , α , β , γ and δ . We wish to establish the implication $\llbracket \Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21})) \rrbracket \supset \llbracket \mathcal{B}(S) \rrbracket$. The dependency graph of $\mathcal{B}(P_1)$, $\mathcal{B}(P_2)$, $\mathcal{B}(M_{12})$, and $\mathcal{B}(M_{21})$, relative to $\mathcal{B}(S)$, is shown in Fig. 4.6.6.

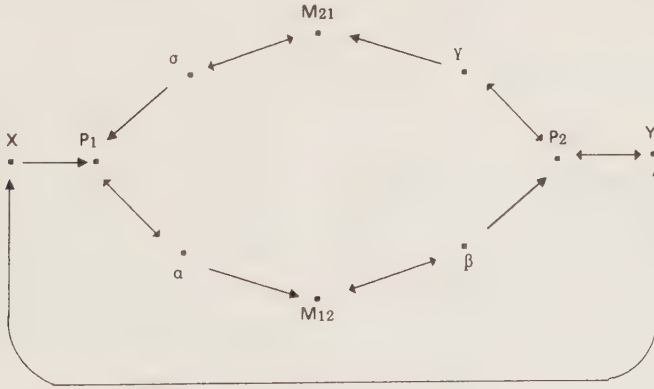


Fig. 4.6.6. Dependency graph of $\mathcal{B}(P_1)$, $\mathcal{B}(P_2)$, $\mathcal{B}(M_{12})$, and $\mathcal{B}(M_{21})$ relative to $\mathcal{B}(S)$,

The single arrows in Fig. 4.6.6 correspond to buffered event names. For example, the event name α is input buffered in $\mathcal{B}(M_{12})$. By removing the buffer buf_α from $\mathcal{B}(M_{12})$, the buffer buf_β from $\mathcal{B}(P_2)$ and the buffer buf_δ from $\mathcal{B}(P_1)$, we obtain BTGs $\mathcal{B}(P_1)'$, $\mathcal{B}(P_2)'$, $\mathcal{B}(M_{12})'$, and $\mathcal{B}(M_{21})'$. The dependency graph of $\mathcal{B}(P_1)'$, $\mathcal{B}(P_2)'$, $\mathcal{B}(M_{12})'$, and $\mathcal{B}(M_{21})'$ relative to $\mathcal{B}(S)$ is shown in Fig. 4.6.7.

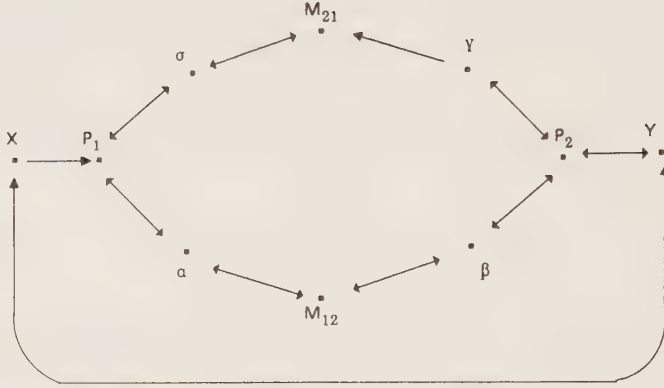


Fig. 4.6.7. Dependency graph of $\mathcal{B}(P_1)'$, $\mathcal{B}(P_2)'$, $\mathcal{B}(M_{12})'$, and $\mathcal{B}(M_{21})'$ relative to $\mathcal{B}(S)$,

By inspection, we note that for each cycle in $\mathcal{G}'$ of length greater than 2 which visits no node twice, there is a corresponding cycle in $\mathcal{G}$. In section 5.3, we shall also prove that the removed buffers are live. Therefore the implication $\llbracket \Pi(\mathcal{B}(P_1)', \mathcal{B}(P_2)', \mathcal{B}(M_{12})', \mathcal{B}(M_{21})') \rrbracket \supset \llbracket \mathcal{B}(S) \rrbracket$ is equivalent to the original one.

Note that we did not also remove the buffer buf_γ , because then $\mathcal{G}'$ would have contained the cycle

$$X \ \mathcal{B}(P_1) \ \delta \ \mathcal{B}(M_{21}) \ \gamma \ \mathcal{B}(P_2) \ Y \ X$$

which is not a cycle in $\mathcal{G}$.

□

4.7 Proofs from Chapter 4

Proof of theorem 4.4.2.

Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be a set of trace generators. We shall prove that

$$\llbracket \Pi(\mathcal{A}_1, \dots, \mathcal{A}_k) \rrbracket = \bigwedge_{i=1}^k \llbracket \mathcal{A}_i \rrbracket$$

According to definition 4.2.2, the theorem follows if we can establish that for each $q \in (E(\llbracket \Pi(\mathcal{A}_1, \dots, \mathcal{A}_m) \rrbracket))^\dagger$, it holds that

$$q \in Q(\llbracket \Pi(\mathcal{A}_1, \dots, \mathcal{A}_m) \rrbracket) \quad \text{iff} \quad \text{for each } i, q \upharpoonright E(\llbracket \mathcal{A}_i \rrbracket) \in Q(\llbracket \mathcal{A}_i \rrbracket).$$

We shall set out to prove this claim.

We first observe is that a state of $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_m)$ is a tuple $\bar{\xi} = \langle \bar{\xi}_1, \dots, \bar{\xi}_m \rangle$ where each $\bar{\xi}_i$ is a state of $\mathcal{A}_i$. We also note that the transitions of $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_m)$ are exactly those transitions that can be obtained by the following rules:

1) For silent transitions

$$\frac{\bar{\xi}_i \xrightarrow{\tau} \bar{\xi}'_i}{\langle \bar{\xi}_1, \dots, \bar{\xi}_i, \dots, \bar{\xi}_k \rangle \xrightarrow{\tau} \langle \bar{\xi}_1, \dots, \bar{\xi}'_i, \dots, \bar{\xi}_k \rangle}$$

i.e., transitions labeled by τ only concern one component $\bar{\xi}_i$ of $\bar{\xi}$.

2) If e is an event name of the trace generators $\mathcal{A}_{i_1}, \dots, \mathcal{A}_{i_m}$, then transitions labeled by e are obtained by the rule

$$\frac{\bar{\xi}_{i_1} \xrightarrow{e(d)} \bar{\xi}'_{i_1} \quad , \quad \dots \quad , \quad \bar{\xi}_{i_m} \xrightarrow{e(d)} \bar{\xi}'_{i_m}}{\langle \bar{\xi}_1, \dots, \bar{\xi}_{i_1}, \dots, \bar{\xi}_{i_k}, \dots, \bar{\xi}_k \rangle \xrightarrow{e(d)} \langle \bar{\xi}_1, \dots, \bar{\xi}'_{i_1}, \dots, \bar{\xi}'_{i_k}, \dots, \bar{\xi}_k \rangle}$$

i.e., transitions with communication event e involve the components for which e is in the set of events.

We can now use a technique similar to that in the proof of the composition rule of theorem 3.3.2 to prove the claim. We shall prove each direction separately.

First assume that q is the sequence of communication events in a computation C of $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_m)$

$$C = \langle \bar{\xi}_1^0, \dots, \bar{\xi}_k^0 \rangle \xrightarrow{e^1} \langle \bar{\xi}_1^1, \dots, \bar{\xi}_k^1 \rangle \xrightarrow{e^2} \dots \xrightarrow{e^n} \langle \bar{\xi}_1^n, \dots, \bar{\xi}_k^n \rangle \xrightarrow{e^{n+1}} \dots$$

We shall construct a computation C_i of $\mathcal{A}_i$ in which the sequence of communication events is $q[E(\llbracket \mathcal{A}_i \rrbracket)]$ by replacing each transition

$$\langle \bar{\xi}_1^{n-1}, \dots, \bar{\xi}_i^{n-1}, \dots, \bar{\xi}_k^{n-1} \rangle \xrightarrow{e^n} \langle \bar{\xi}_1^n, \dots, \bar{\xi}_i^n, \dots, \bar{\xi}_k^n \rangle$$

in C by $\bar{\xi}_i^{n-1} \xrightarrow{e^n[E(\llbracket \mathcal{A}_i \rrbracket)]} \bar{\xi}_i^n$. Note that this always yields a transition of $\mathcal{A}_i$.

The result of this transformation is a sequence of transitions of $\mathcal{A}_i$ whose sequence of communication events is $q[E(\llbracket \mathcal{A}_i \rrbracket)]$. It remains to prove that the sequence is indeed a computation of $\mathcal{A}_i$. The conditions for being a computation are checked below.

- (A) *Initialization.* $\bar{\xi}_i^0$ must be the initial state of $\mathcal{A}_i$, since it is the i th component of $\bar{\xi}^0$.
- (B) *State-to-state sequencing.* This condition follows from the transformation from C to C_i .
- (C) *Guarantee.* Follows from the fact that each guarantee-condition of $\mathcal{F}_i$ is also a guarantee-condition of $\mathcal{F}$.
- (D) *Fairness.* Follows in the same way as condition (C).

This concludes the proof in one direction. Now we turn to the other direction of the theorem. Namely, assume that for each i there is a computation C_i

$$\bar{\xi}_i^0 \xrightarrow{e_i^1} \bar{\xi}_i^1 \xrightarrow{e_i^2} \dots \xrightarrow{e_i^n} \bar{\xi}_i^n \xrightarrow{e_i^{n+1}} \dots$$

in which the sequence of communication events is $q[E(\llbracket \mathcal{A}_i \rrbracket)]$. In the same way as in the proof of theorem 3.3.2, we use lemma 3.6.1 to conclude we conclude that there is a partial computation of $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_m)$ in which each component $\mathcal{A}_i$ performs the sequence of transitions C_i .

We must verify that C is really a computation of $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_m)$. We treat the two extra requirements:

- (C) *Guarantee*. This requirement follows from the fact each guarantee requirement of $\mathcal{A}_i$ is also a guarantee requirement of $\Pi(\mathcal{A}_1, \dots, \mathcal{A}_m)$.
- (D) *Fairness*. Follows similarly as guarantee. Note that by our definition of infinitely often, it does not matter whether the last state of a finite computation is repeated an infinite number of times.

□

Proof of theorem 4.5.3.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two trace specifications. Assume that $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$, and

- (1) $\mathcal{A}_1$ is deadlock-free,
- (2) $\mathcal{A}_2$ is deterministic, and
- (3) for each guarantee requirement $\phi_i \hookrightarrow \psi_i$ of $\mathcal{A}_1$, we have $(\phi_i) \xrightarrow{R} \circ (\psi_i ; \phi_i)$,

We must show that there exists a simulation relation R , in the set-theoretic sense, between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$ such that the premises of theorem 4.4.6 hold.

Premise (1), that $E_2 \subseteq E_1$, follows by the fact that $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$ implies $E(\llbracket \mathcal{A}_2 \rrbracket) \subseteq E(\llbracket \mathcal{A}_1 \rrbracket)$ according to definition 4.2.2.

To establish premise (2), we define the simulation relation R between the states of $\mathcal{A}_1$ and $\mathcal{A}_2$. Define $R(\bar{\xi}_1, \bar{\xi}_2)$ to be true iff there is a $q \in (E(\llbracket \mathcal{A}_1 \rrbracket))^*$, and finite partial computations C_1 of $\mathcal{A}_1$ and C_2 of $\mathcal{A}_2$, such that

- (i) $E(C_1) = q$ and $E(C_2) = q[E(\llbracket \mathcal{A}_2 \rrbracket)]$,
- (ii) the last state of C_1 is $\bar{\xi}_1$, and
- (iii) the last state of C_2 is $\bar{\xi}_2$.

To verify that R is a simulation relation, we check the conditions of definition 4.4.2.

- (1) Consider initial states $\bar{\xi}_1^0$ and $\bar{\xi}_2^0$ of $\mathcal{A}_1$ and $\mathcal{A}_2$. By putting $q = \langle \rangle$, $C_1 = \bar{\xi}_1^0$ and $C_2 = \bar{\xi}_2^0$ in the above definition of R , we conclude that $R(\bar{\xi}_1^0, \bar{\xi}_2^0)$ holds.

- (2) Assume that $R(\bar{\xi}_1, \bar{\xi}_2)$ and that $\bar{\xi}_1 \xrightarrow{e(d)} \bar{\xi}'_1$ is a transition of $\mathcal{A}_1$. We must show that there is a state $\bar{\xi}'_2$ of $\mathcal{A}_2$ such that $R(\bar{\xi}'_1, \bar{\xi}'_2)$ holds and such that $\bar{\xi}_2 \xrightarrow{e(d) \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)} \bar{\xi}'_2$ is a transition of $\mathcal{A}'_2$.

By the definition of R , the states $\bar{\xi}_1$ and $\bar{\xi}_2$ are the last states of two partial computations C_1 and C_2 . Let $q = E(C_1)$. Extend C_1 by the transition $\bar{\xi}_1 \xrightarrow{e(d)} \bar{\xi}'_1$ to get C'_1 . Since $\mathcal{A}_1$ is deadlock-free, C'_1 can be extended to a computation of $\mathcal{A}_1$. Let q'' be the sequence of communication events in this computation. Since $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$, there exists a computation of $\mathcal{A}_2$ in which the sequence of communication events is $q'' \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)$. Thus there is a partial computation C'_2 of $\mathcal{A}_2$ such that $E(C'_2) = (q \upharpoonright e(d)) \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)$. Since $\mathcal{A}_2$ is deterministic, C_2 must be a prefix of C'_2 . Let $\bar{\xi}'_2$ be the last state of C'_2 . It follows that $R(\bar{\xi}'_1, \bar{\xi}'_2)$, and that hence C'_2 extends C_2 by the transition $\bar{\xi}_2 \xrightarrow{e(d) \upharpoonright E(\llbracket \mathcal{A}_2 \rrbracket)} \bar{\xi}'_2$.

We next verify that the conditions (2) in theorem 4.4.6 hold for sequences of R -transitions. In this proof, we shall first establish the following

Lemma: If C is a sequence of joint R -transitions that satisfy $\mathcal{F}_1$, then C is a suffix of a sequence of joint R -transition that satisfy $\mathcal{F}_1$, and in which the components of the first joint R -state are initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Proof. By the construction of R , C is a suffix of a sequence C' of joint R -transition, in which the components of the first joint R -state are initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$. We must only verify that C' satisfies $\mathcal{F}_1$.

If $\phi \hookrightarrow \psi \in \mathcal{F}_1$, and ϕ holds for some joint R -state in C' , then either

- (1) ϕ holds in C , whence there is a ψ -transition in C and hence in C' , since C satisfies $\mathcal{F}_1$.
- (2) ϕ holds before C . If ψ never holds, then by condition (3) in the statement of the theorem ϕ holds throughout C' , and hence also in C , which by the previous case leads to a contradiction.

If $\langle \phi, \psi \rangle \in \mathcal{F}_1$, then ϕ holds infinitely often in C' if and only if ϕ holds infinitely often in C . It follows that C' satisfies this fairness requirement.

□

We shall now prove that the two requirements on R in theorem 4.4.6 are satisfied

- (i) Let $\phi \hookrightarrow \psi$ be a guarantee requirement in $\mathcal{F}_2$. Let C be a sequence of joint R -transitions which satisfies the requirements in $\mathcal{F}_1$, and whose first joint R -state satisfies ϕ . We must show that C contains a joint R -transition that satisfies ψ .

By the previous lemma, C is a suffix of a sequence C' of joint R -transitions that satisfy $\mathcal{F}_1$, and in which the components of the first joint R -state are

initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$. The sequence of first components of C' is a computation of $\mathcal{A}_1$, since it satisfies $\mathcal{F}_1$. Since $\llbracket \mathcal{A}_1 \rrbracket \supset \llbracket \mathcal{A}_2 \rrbracket$, and $\mathcal{A}_2$ is deterministic, the sequence of second components must be a computation of $\mathcal{A}_2$, and therefore ψ must hold sometime after ϕ .

(ii) The proof is analogous to case (i).

□

Proof of theorem 4.5.4.

Let R be a simulation relation between the states of the two trace specifications $\mathcal{A}_1$ and $\mathcal{A}_2$. Assume that

- (1) for each guarantee requirement $\phi_i \hookrightarrow \psi_i$ of $\mathcal{A}_1$, we have $(\phi_i) \xrightarrow{R} \circ (\psi_i ; \phi_i)$
- (2) $\mathcal{A}_1$ and $\mathcal{A}_2$ have only finitely many liveness requirements

We shall prove that the proof rules for auxiliary formulas in section 4.4.3 are semantically complete. The proof is adapted from Francez ([Fr 86]).

In the proof we do not distinguish between formulas and sets of states or transitions. Thus, the word formula refers to a set of states or transitions, regardless of whether the set can be expressed in any language. Analogously, we allow the components P, U , etc. in $(P) \xrightarrow{R} \diamond (T ; U)$ to be sets, regardless whether they can be expressed in any language. The meaning of $(P) \xrightarrow{R} \diamond (T ; U)$ is analogous to the case when P, Q , etc. are formulas.

Assume that $(P) \xrightarrow{R} \diamond (T ; U)$ holds. We shall prove that there exists a well-founded set $(W, <)$ and a set of states $\chi(w)$ parameterized by elements in W , such that

$$P \supset (U \vee (\exists w) \chi(w))$$

$$(\chi(w)) \xrightarrow{R} \diamond (T ; (\exists w' < w) \chi(w') \vee U) \quad \text{for all } w \in W$$

holds. We shall also prove that for each $w \in W$, the second formula follows from an application of the guarantee rule or the fairness rule, from premises that are true. In the case where the fairness rule is used, a premise contains another formula of the form $(P) \xrightarrow{R} \diamond (T ; U)$, but this time $\mathcal{F}'$ has one fairness requirement less than $\mathcal{F}$, as so this process terminates in a finite number of steps.

We assume that each liveness requirements $\phi_h \hookrightarrow \psi_h$ or $\langle \phi_h, \psi_h \rangle$ in $\mathcal{F}$ is indexed by a unique index h from a finite index set I .

Let $\bar{\xi}^0$ be a joint R -state that satisfies P . We organize all sequences of joint R -transitions that start in $\bar{\xi}^0$ into a tree $T(\bar{\xi}^0)$ in which the nodes are occurrences of joint R -states. If $\bar{\xi}$ is a node in $T(\bar{\xi}^0)$, and $\bar{\xi} \xrightarrow{e(d)} \bar{\xi}'$ is a joint

R -transition, then $T(\bar{\xi}^0)$ has an arc labeled by $e(d)$ leading from $\bar{\xi}$ to the child $\bar{\xi}'$. In the following, we will simply use the words transition and state for joint R -transition and joint R -state.

The idea of the proof is to collapse families of nodes of $T(\bar{\xi}^0)$ into single nodes to obtain another tree $T^*(\bar{\xi}^0)$. The tree $T^*(\bar{\xi}^0)$ is well-founded, i.e., it contains no infinite paths. We can then assign a *rank* to each node of $T^*(\bar{\xi}^0)$, which is also assigned to each of the nodes of the corresponding family of nodes of $T(\bar{\xi}^0)$. From this assignment of ranks we obtain the parameterized state formula $\chi(w)$.

Consider a node $\bar{\xi}$ in $T(\bar{\xi}^0)$, and an index $h \in I$ of a liveness requirement $\phi_h \hookrightarrow \psi_h$ or $\langle \phi_h, \psi_h \rangle$ in $\mathcal{F}$. Define $CONE_h(\bar{\xi})$ as the set of all nodes in $T(\bar{\xi}^0)$ that reside on a path, which starts in $\bar{\xi}$ and does not contain any ψ_h -state, U -state, ψ_h -transition, or T -transition. Note that if $\bar{\xi}$ itself is a U -state or ψ -state, then $CONE_h(\bar{\xi})$ is empty. The index h is called the *directive* of the cone.

We shall now define the families of states that define the nodes of the collapsed tree $T^*(\bar{\xi}^0)$. If $\bar{\xi}^0$ is a U -state, then the tree $T^*(\bar{\xi}^0)$ consists of the only family $\{\bar{\xi}^0\}$. Otherwise, we define the families inductively level by level, starting with families at level 0, and proceeding upwards. At each level i , we define a set of *distinguished nodes*. For each distinguished node $\bar{\xi}_i$, we define a family of nodes $A(\bar{\xi}_i)$, a sequence $\pi(\bar{\xi}_i)$ which contains all indices in I in some order, and a set of descendant distinguished nodes at level $i + 1$.

Base Case: Let the distinguished node $\bar{\xi}_0$ at level 0 be the root of $T(\bar{\xi}^0)$. Let $\pi(\bar{\xi}_0)$ contain the indices of I in any order

Inductive Step: Assume that $\bar{\xi}_i$ is a distinguished node at level i .

Consider the set of indices $h \in I$ of liveness requirements such that either

- (i) $\phi_h \hookrightarrow \psi_h \in \mathcal{F}$ and $\bar{\xi}_i$ is a ϕ_h -state, or
- (ii) $\langle \phi_h, \psi_h \rangle \in \mathcal{F}$ and $CONE_h(\bar{\xi}_i)$ contains a state or a transition which satisfies ϕ_h .

Note that there must be at least one $h \in I$ for which (i) or (ii) holds, otherwise the sequence of states may end in $\bar{\xi}^i$ while satisfying $\mathcal{F}$, without having reached a U -state or a T -transition.

Let h' be the first index in $\pi(\bar{\xi}_i)$ for which (i) or (ii) holds. Let $A(\bar{\xi}_i) = CONE_{h'}(\bar{\xi}_i)$, and let any node, which is the first state outside $A(\bar{\xi}_i)$ on a path from $\bar{\xi}_i$, be a distinguished node $\bar{\xi}_{i+1}$ at level $i + 1$, if the node is not a U -state, and the transition leading to the node is not a T -transition. To obtain $\pi(\bar{\xi}_{i+1})$, put index h' at position i (or last) in $\pi(\bar{\xi}_i)$.

Before defining the tree $T^*(\bar{\xi}^0)$, we state the following lemma:

Lemma (Cone-chain): Consider the hierarchy of families $A(\bar{\xi}_i)$. There is no infinite sequence of families $A(\bar{\xi}_i) = \text{CONE}_{h_i}(\bar{\xi}_i)$ which are descendants of each other, such that $\langle \bar{\xi}_i \rangle_{i=0}^\infty$ resides on an infinite path in $T(\bar{\xi}^0)$ on which T or U never holds.

Proof. Assume that there exists such a path. By the assumptions of the theorem, it must violate some liveness requirement of $\mathcal{F}$, since T or U never holds. There are two cases:

- (a) The violated liveness requirement is of form $\phi_h \hookrightarrow \psi_h$. At some point on the path, ϕ_h must hold, but from then on ψ_h never holds. By the condition of the theorem, ϕ_h holds continuously from then on. By the treatment of the sequence $\pi(\bar{\xi})$ of indices in I , eventually h will be the first index to be considered under case (2) above. Thus there is a j such that $A(\bar{\xi}_j) = \text{CONE}_h(\bar{\xi}_j)$. Since beyond that point, neither ψ_h nor T nor U holds, the rest of the path will stay in $A(\bar{\xi}_j)$, contradicting the assumption that the path passes infinitely many families $A(\bar{\xi}_i)$.
- (b) The violated liveness requirement is of form $\langle \phi_h, \psi_h \rangle$. From some point at the path, ϕ_h must hold infinitely often, but from then on ψ_h never holds. By the treatment of the sequence $\pi(\bar{\xi})$ of indices in I , eventually h will be the first index to be considered under case (2) above. Thus there is a j such that $A(\bar{\xi}_j) = \text{CONE}_h(\bar{\xi}_j)$. Since beyond that point, neither ψ_h nor T nor U holds, the rest of the path will stay in $A(\bar{\xi}_j)$, contradicting the assumption that the path passes infinitely many families $A(\bar{\xi}_i)$.

□

We can now construct the tree $T^*(\bar{\xi}^0)$. Its nodes are the families A_i , and its edges are the edges of $T(\bar{\xi}^0)$ that go between families. By the previous lemma, the tree $T^*(\bar{\xi}^0)$ is well-founded, i.e., contains no infinite paths. In order to avoid dependency on the particular initial state, combine all trees $T^*(\bar{\xi}_0)$ for which $\bar{\xi}_0$ satisfies P into one tree T^* with a new root, which has all trees $T^*(\bar{\xi}_0)$ as descendants.

We now assign a rank to each node of T^* in a standard manner, using ordinals: leaves get rank 0 and other nodes get the least rank which is larger than the rank of each of their descendants. Call this rank ρ the *basic rank*. A problem will arise later if two nodes with same basic rank are cones with different directives in I . We therefore change the rank: Assume that the indices in I are linearly ordered. Replace the rank ρ of a cone with directive h by the pair $w = \langle \rho, h \rangle$. Let W be the set of these tuples, and $<$ be the lexicographic ordering between them. Since I is at most countable, we get a new rank w , such that two nodes with the same rank have the same directive.

We now define $\chi(w)$ with $w \in W$. Let $\chi(w)$ be true if and only if $\bar{\xi}$ belongs to a family with rank w . We shall now verify the claims done at the beginning of the proof.

First note that each state which satisfies P is a descendant of the root of the tree T^* . Each such descendant has a rank, unless it is a U -state. Therefore the formula

$$P \supset (U \vee (\exists w)\chi(w))$$

holds. We shall also establish that each premise of the chain rule

$$(\chi(w)) \xrightarrow[R]{\mathcal{F}} \Diamond (T ; (\exists w' \prec w)\chi(w') \vee U)$$

can be proven either by a guarantee rule or by a fairness rule for each $w \in W$. Consider a $w \in W$. Assume that $\chi(w)$ holds in the state $\bar{\xi}$, i.e., that $\bar{\xi}$ belongs to a cone $A(\bar{\xi}_i) = CONE_h(\bar{\xi}_i)$ with rank w . There are two cases, depending on the nature of h

- (a) h is the index of a guarantee requirement $\phi_h \hookrightarrow \psi_h$. The three premises of the guarantee-rule are verified as follows:

(1) $(\chi(w)) \xrightarrow[R]{\circ} (T ; (\exists w' \preceq w)\chi(w') \vee U)$ is true, since a transitions from a $\chi(w)$ -state $\bar{\xi}$ either stays within in $A(\bar{\xi}_i)$, or leaves $A(\bar{\xi}_i)$ by a T -transition, a ψ_h -transition which decrease w , or by going to a state which satisfies U .

(2) $(\chi(w)) \xrightarrow[R]{\circ} (\psi_h \supset T ; (\exists w' \prec w)\chi(w') \vee U)$ is true, since each ψ_h -transition leaves $A(\bar{\xi}_i)$. Then either we use a T -transition or get to a U -state, or to a state with lower rank.

(3) $\chi(w) \supset (\phi_h \vee U)$ is true, since all states in $A(\bar{\xi}_i)$ are ϕ_h -states.

- (b) h is the index of a fairness requirement $\langle \phi_h, \psi_h \rangle$. The first two premises of the fairness rule are verified in the same way as in case (a). We must verify the third premise. If ϕ_h and ψ_h are both transition formulas, this becomes

$$(\chi(w)) \xrightarrow[R]{\mathcal{F}'} \Diamond (\phi_h \vee T ; (\exists w' \prec w)\chi(w') \vee U)$$

To prove this by contradiction, assume that from $\bar{\xi}$ there is a sequence of R -transitions in which neither $\phi_h \vee T$ nor $U \vee (\exists w' \prec w)\chi(w')$ holds, and which satisfies $\mathcal{F}'$. Such a sequence stays within $A(\bar{\xi}_i)$. If the sequence contains no ϕ_h -state, then it satisfies $\mathcal{F}$ as well. This contradicts the premise of the theorem, since we now have a sequence which satisfies $\mathcal{F}$, without containing any T -transition or U -state.

□

Proof of theorem 4.6.12.

Let $\mathcal{B}_1, \dots, \mathcal{B}_k$ and $\mathcal{B}'_1, \dots, \mathcal{B}'_k$ be buffered trace generators as in the formulation of the theorem, i.e., $\mathcal{B}'_i$ is obtained by removing buffers from $\mathcal{B}_i$. Assume that the conditions of the theorem are satisfied. Let $E(\llbracket \mathcal{B} \rrbracket)$ be the set of events of $\mathcal{B}$. For each sequence $q \in E(\llbracket \mathcal{B} \rrbracket)^\dagger$, we must show that there is a computation C of $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ such that $E(C) \upharpoonright E(\llbracket \mathcal{B} \rrbracket) = q$ if and only if there is a computation C' of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$ such that $E(C') \upharpoonright E(\llbracket \mathcal{B} \rrbracket) = q$.

We first consider the structure of computations of $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ and of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$. A state of $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ is a tuple $\langle \bar{\xi}_1, \dots, \bar{\xi}_m, buf_1, \dots, buf_l \rangle$, in which the components $\bar{\xi}_1, \dots, \bar{\xi}_k$ are the states of the trace generator $\mathcal{B}_1, \dots, \mathcal{B}_k$ with all buffers removed, and $buf_1, \dots, buf_l$ are the states of all buffers of $\mathcal{B}_1, \dots, \mathcal{B}_k$. The components of a state of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$ are obtained by removing some of the buffers components from $\langle \bar{\xi}_1, \dots, \bar{\xi}_m, buf_1, \dots, buf_l \rangle$, but we shall represent a state of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$ by the same number of components as the state of $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ by the convention that a removed buffer is always empty.

A transition of $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ is the result of simultaneous transitions in a number of components. Just as in the proof of theorem 3.3.2, we shall represent a transition by the set of participating component transitions. This is called a set-representation of a transition. We also use set-representations of transitions of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$. If in a transition of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$ the component $\bar{\xi}_i$ performs an event e , and buf_e is removed from $\mathcal{B}_i$, then the set-representation of that transition also includes a component transition of (the removed) buf_e which inserts an element into buf_e and a component transition which removes an element from buf_e .

Assume now that C' is a computation of $\Pi(\mathcal{B}'_1, \dots, \mathcal{B}'_k)$. We shall prove that there exists a computation C of $\Pi(\mathcal{B}_1, \dots, \mathcal{B}_k)$ with the same sequence of events of $E(\llbracket \mathcal{B} \rrbracket)$. Let $\mathfrak{S}$ be the multiset of component transitions that occur in C' , i.e., we distinguish between different occurrences of the same transition. Introduce the following notation:

$t_{\bar{\xi}_i}^n$ is the n th occurrence of a transition of component $\bar{\xi}_i$ in C .

$t_{in_j}^n$ is the n th transition of buf_j in which an element is inserted into buf_j .

$t_{out_j}^n$ is the n th transition of buf_j in which an element is removed from buf_j .

We define a preorder relation $\leq$ on $\mathfrak{S}$ as the transitive closure of the following relations. (We use $t \equiv t'$ to denote $t \leq t' \leq t$)

- (1) $t_{\bar{\xi}_i}^m \equiv t_{out_j}^n$ if buf_j is an input buffer of B_i , and $t_{\bar{\xi}_i}^m$ is the n th transition in B_i that removes an element from buf_j .

- (2) $t_{\xi_i}^m \equiv t_{in_j}^n$, if buf_j is an output buffer of B_i , and $t_{\xi_i}^m$ is the m th transition in B_i that inserts an element into buf_j .
- (3) $t \equiv t'$ if both t and t' exhibit the n th communication event e in C' for some n .
- (4) $t \preceq t'$ if the component transition t performs a communication event that corresponds to the n th communication event of $\llbracket B \rrbracket$ in C' , and t' performs a communication event that corresponds to the n th or $(n+1)$ st communication event of $\llbracket B \rrbracket$ in C' .
- (5) $t_{\xi_i}^n \preceq t_{\xi_i}^{n+1}$, i.e., the transitions of each B_i occurs in the same order as in C' .
- (6) $t_{out_j}^n \preceq t_{out_j}^{n+1}$, $t_{in_j}^n \preceq t_{out_j}^n$, and $t_{in_j}^n \preceq t_{in_j}^{n+1}$, These constraints constrain the occurrence of events of each buffer buf_j .

For each buffer buf_j that is removed when obtaining B'_i from B_i , we also have

- (7) $t_{out_j}^n \preceq t_{in_j}^n$, if the buffer buf_j is removed to obtain $B'_1, \dots, B'_k$.

The preorder relation defined by (1) - (7) induces a set of equivalence classes. It is not difficult to verify that each equivalence class is a set-representation of a transition in C' .

In order to transform C' to a computation C of $\Pi(B_1, \dots, B_k)$ we remove condition (7) from the definition of $\preceq$. We claim that the equivalence classes will now be set-representations of transitions of $\Pi(B_1, \dots, B_k)$. If the claim is true, then we simply take C as a linear sequence of these set-representations.

To verify the claim, assume that an equivalence class induced by requirements (1) - (6) does not correspond to the set-representation of a transition of $\Pi(B_1, \dots, B_k)$. In that case there is a cycle

$$t \preceq \dots \preceq t' \preceq \dots \preceq t$$

of component transitions that contain more component transitions than a transition of $\Pi(B_1, \dots, B_k)$. This cycle must be contained in the set-representation of a transition in C' . It follows that two component transitions must be related according to requirement (7) above, which is a contradiction.

We now verify the theorem in the other direction. Assume that C is a computation of $\Pi(B_1, \dots, B_k)$. We shall construct a computation C' of $\Pi(B'_1, \dots, B'_k)$ with the same sequence of events in $E(\llbracket B \rrbracket)$.

We shall perform the proof in the preceding case backwards. That is, let $\mathfrak{F}$ be the set of occurrences of component transitions in C . Define a preorder on $\mathfrak{F}$ by clauses corresponding to (1) - (6) above, but using C instead of C' in the definitions. The equivalence classes of component transitions are set-representations of transitions in C ,

We now add requirement (7) to the preorder. We claim that the resulting set of equivalence classes are transitions of $\Pi(B'_1, \dots, B'_k)$. If the claim is true, we can take C' as a linear sequence of equivalence classes.

To prove the claim, assume that there is an equivalence class of transitions obtained from (1) - (7), which contains more component transitions than a transition of $\Pi(B'_1, \dots, B'_k)$. Then the equivalence class contains a cycle of component transitions

$$t \preceq \dots \preceq t' \preceq \dots \preceq t \quad (*)$$

where each pair is related by one of the constraints (1) - (7), and which contains more component transitions than a transition of $\Pi(B'_1, \dots, B'_k)$. We can assume that t , t' , and $(*)$ are chosen so that $(*)$ is as short as possible. Since the constraints (1) - (6) allow the construction of a computation C , at least one pair must be ordered by (7). Now shift the cycle so that the pair satisfying (7) occurs as the first one, i.e., $(*)$ becomes

$$t_1 \preceq t_2 \dots \preceq t_k \preceq t_1 \quad (**)$$

where $t_1 \preceq t_2$ follows from constraint (7). We assume that $t_1 = t_{out_j}^n$ and $t_2 = t_{in_j}^n$ for an output buffer buf_j of B_i . (the case of an input buffer is treated analogously). We now use requirement (1) of the theorem, which implies that in the dependency graph of $B'_1, \dots, B'_k$ relative to B there is no path from B_i to e , where $buf_j = buf_e$. This implies that the transitions in $(**)$ that precede t_1 must be an input transition of buf_j followed by output transitions of buf_j . Thus the sequence $(**)$ is of the form

$$t_{out_j}^n \preceq t_{in_j}^n \preceq \dots \preceq t_{in_j}^m \preceq t_{out_j}^m \preceq \dots \preceq t_{out_j}^{n-1} \preceq t_{out_j}^n$$

This sequence can be replaced by the shorter cycle

$$t_{in_j}^n \preceq \dots \preceq t_{in_j}^m \preceq \dots \preceq t_{in_j}^{n-1} \preceq t_{in_j}^n$$

in which the output transitions from buf_j are deleted. By the assumption that $(*)$ is as short as possible, we conclude that the transitions in this sequence are in one transition of $\Pi(B'_1, \dots, B'_k)$. But due to the way transitions of $\Pi(B'_1, \dots, B'_k)$ are constructed, we conclude that $(*)$ must also be in one transition of $\Pi(B'_1, \dots, B'_k)$. Hence we have derived a contradiction. $\square$

Proof of theorem 4.6.13.

Assume first that e is input buffered in both B_1 and B_2 . Since $\llbracket B'_1 \rrbracket \supset \llbracket B_2 \rrbracket$, for each computation C'_1 of B'_1 with sequence of events q' there is a computation C'_2 of B_2 with the sequence of events $q[E(\llbracket B_2 \rrbracket)]$.

Now consider a computation C_1 of B_1 . In this computation, the component B'_1 performs a computation, and the buffer buf_e performs a sequence of transitions. For each transition of B'_1 which retrieves an element from buf_e there is

an earlier corresponding transition of buf_e in which this element is inserted into buf_e . If buf_e is not live, there may be additional insertions into buf_e that follow the insertion that corresponds to the last removal from buf_e .

We can construct a computation C_2 of B_2 by transforming C'_2 as follows: Each transition which inserts an element into buf_e of B_2 , which in C'_2 corresponds to the transition in C_1 that removes the element from buf_e is moved further to the start of the computation so that it corresponds to the transition that inserts the corresponding element into buf_e in C_1 . If buf_e is not live in B_1 , we also add the extra inserting transitions that have no removing transitions.

The only thing that is still to be checked is a possible buffer guarantee or fairness condition for e in B_2 . In this case, the theorem requires that buf_e be live in B_1 . Therefore no extra inserting transitions are added when C_2 is formed from C'_2 . and it follows that each inserting transition in C_2 has a corresponding removing transition in C_2 .

The case when e is output buffered is treated analogously. $\square$

5 Applications

In this chapter we illustrate the specification and verification by applying it to three examples. In section 5.1, we specify and verify the counterexample of Brock and Ackermann ([BA 81]). In section 5.2, we specify and verify a sliding window protocol for ensuring correct transfer of messages across a faulty medium. The window mechanism appears in the link protocol HDLC. In section 5.3, we specify and verify an algorithm, due to Thomas ([Th 79]), that ensures consistency of updates in a distributed database.

5.1 The Brock-Ackermann Counterexample

Kahn ([Ka 74]) has presented a model for deterministic Kahn-networks, which represents a network as a function from histories of its input channels to histories of its output channels. This model is elegant, but unfortunately it does not generalize directly to nondeterministic networks. The obvious generalization is to represent a nondeterministic network as a relation between the histories of its channels. Brock and Ackermann ([BA 81]) have demonstrated that this model is not compositional. They presented a nondeterministic Kahn-network, and then replaced one of its components by another component with the same history relation. The resulting network had a different history relation than the original network. This demonstrates that the history relations of components sometimes do not contain enough information to determine the history relation of a larger system.

In this section, we shall specify the networks and their components, and verify that the compositions of the components satisfy the specification of the networks. We choose this example because it has also been treated by other methods: Brock and Ackermann ([BA 81]) model this example using the scenario model, and Nguyen et al ([NDGO 86]) verify it using temporal logic with predicates and functions over finite sequences.

5.1.1 Informal Description of the Network

In the example, we consider two networks, T_1 and T_2 , where T_i is composed of the networks S_i and $Plus1$. The network S_i is in its turn composed of the components D_1 , D_2 , $Merge$, and P_i for $i = 1, 2$. Thus the difference between the two networks is the component P_i . The components are connected by FIFO channels, over which they transmit integers. The interconnection structure of the network T_i is shown in Fig. 5.1.1

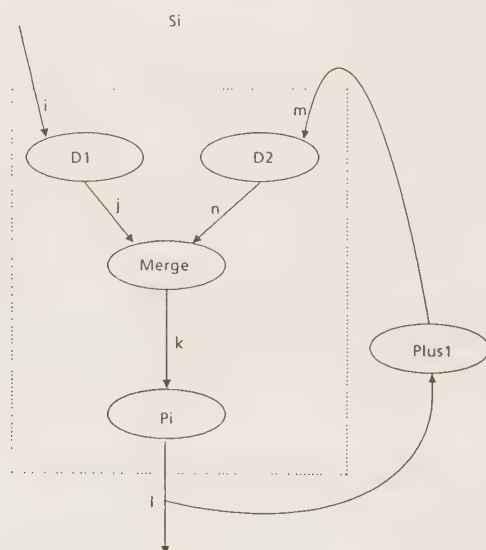


Fig. 5.1.1. Structure of the network T_i .

The point of the example is that S_1 and S_2 are represented by the same history relation, whereas when they are connected with $Plus1$ they yield networks T_1 and T_2 with different history relations. Below is a short description of the behavior of each component:

D_1 receives a datum from i , and transmits it twice on j .

D_2 receives a datum from m , and transmits it twice on n .

$Merge$ merges the data arriving on its input channels and transmits them on the output channel.

P_1 reads a datum from k and transmits it onto l . This operation is repeated once more but thereafter P_1 terminates.

P_2 also reproduces the two first data items received on k onto l , but it first reads two items on k before transmitting anything onto l .

$Plus1$ increments each integer received on its input channels by 1 and transmits the result on m .

When presented with a value a on i and a value b on m , both $S1$ and $S2$ will on channel l produce two values, each of which is either a or b . Thus $S1$ and $S2$ are characterized by identical history relations. However, since $P2$ produces its first output value only after the second output value has been determined, the same holds for $S2$. This difference becomes apparent when adding $Plus1$ to the network, to yield the networks $T1$ and $T2$. When presented with the value a on i , $T1$ will produce any of the sequences $\langle a, a \rangle$, or $\langle a, a + 1 \rangle$ on l , whereas $T2$ can only produce the sequence $\langle a, a \rangle$. The reason for this is that $T2$ has already decided to output a second a before the first a is incremented by $Plus1$ and fed into $S2$. Thus $T1$ and $T2$ are characterized by different history relations.

5.1.2 Specifications of the components.

We specify each component by a buffered trace generator. $D1$ is specified by $\mathcal{B}(D1)$, etc. The event names of each trace generator are the names of the channels that are incident to the component. The parameter v denotes the integer value that is transmitted over the channel. Since values are transmitted via FIFO channels, each event name corresponding to an input channel is provided with the attribute “input buffered.” The states and transitions of each trace generator are given as a state-transition diagram. Circles denote states, and edges denote transitions. The label of an edge denotes the event names of the transition. The initial state is pointed to by an edge without source. Liveness requirements are specified by distinguishing between single and double circles as follows. For each single circle there is a guarantee requirement stating that if the trace generator is in a state represented by a single circle, then at some later point a transition must be taken which leaves that state. States represented by double circles do not have this liveness requirement, and can be thought of as “final states.” For each buffered event name, we specify a buffer fairness requirement. Fig. 5.1.2 shows the buffered trace generators that specify the components.

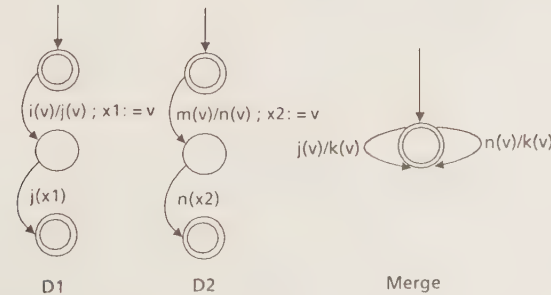


Fig. 5.1.2. Buffered trace generators specifying $D1$, $D2$, and $Merge$.

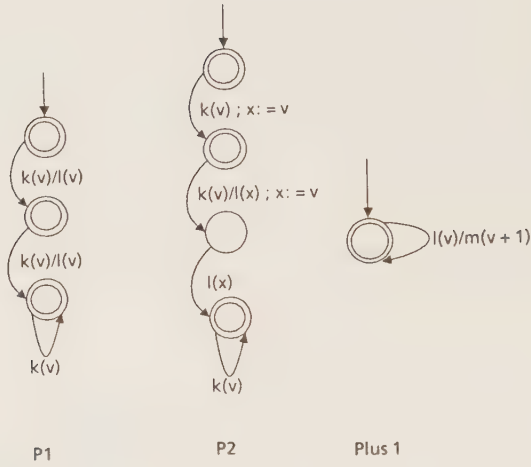


Fig. 5.1.3. Buffered trace generators specifying P_1 , P_2 , and $Plus1$.

5.1.3 Specification of the Network

The networks $S1$ and $S2$ are specified by two buffered trace generators in Fig. 5.1.4. We use the same conventions as in the specification of the components. The event names are i and m which are input buffered, and l .

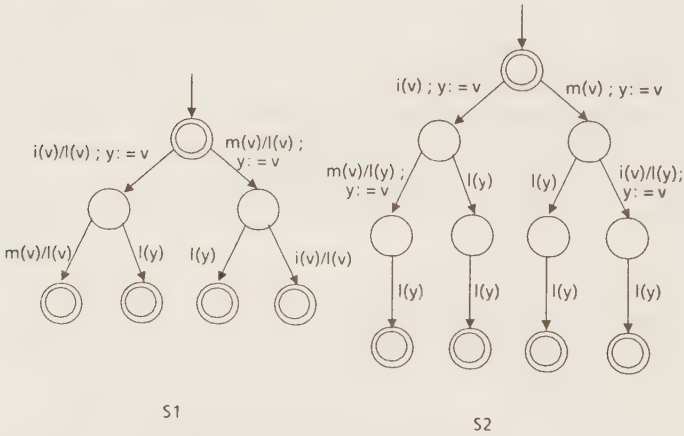


Fig. 5.1.4. The buffered trace generators $S1$ and $S2$.

5.1.4 Verification of $S1$ and $S2$.

To verify that the composition of the components satisfy the specifications of $S1$ and $S2$, respectively, we must prove the implication

$$\llbracket \Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(\text{Merge}), \mathcal{B}(Pi)) \rrbracket \supset \llbracket \mathcal{B}(Si) \rrbracket \quad (*)$$

for $i = 1, 2$. We first form $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(\text{Merge}), \mathcal{B}(Pi))$. In this case, theorem 4.6.12 can be applied to remove the buffers buf_j , buf_n , and buf_k . Note that these buffers are live. The result is the two buffered trace generators in Fig. 5.1.5 and Fig. 5.1.6. The event names of the product is the union of the event names of the components. For clarity, we have only written out those event names which are event names of $\mathcal{B}(Si)$.

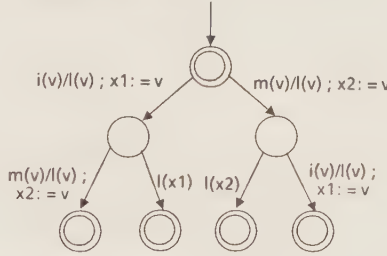


Fig. 5.1.5. $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(\text{Merge}), \mathcal{B}(P1))$.

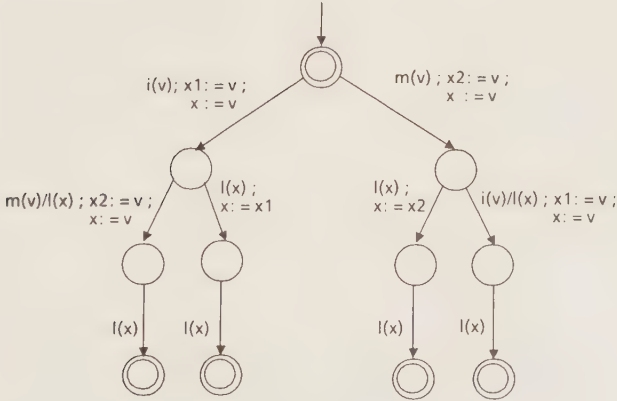


Fig. 5.1.6. $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(\text{Merge}), \mathcal{B}(P2))$.

We shall now establish a simulation between the state of the product of the components specifications, $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(\text{Merge}), \mathcal{B}(Pi))$, and the state of $\mathcal{B}(Si)$ for $i = 1, 2$. We first expand the buffers buf_i and buf_m in both $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(\text{Merge}), \mathcal{B}(Pi))$ and $\mathcal{B}(Si)$. The simulation relation now prescribes the following:

5 Applications

- (1) The state of the input buffers is the same in $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(Merge), \mathcal{B}(Pi))$ and $\mathcal{B}(Si)$ for $i = 1, 2$.
- (2) The state represented by the position in the state-diagram is the same in $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(Merge), \mathcal{B}(Pi))$ and $\mathcal{B}(Si)$ for $i = 1, 2$.
- (3) The variable x in $\mathcal{B}(S1)$ is equal to the variable x_1 in the left subtree of $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(Merge), \mathcal{B}(P1))$, and to the variable x_2 in the right subtree.
- (4) In $\mathcal{B}(S2)$ the variable x corresponds to the variable y in $\Pi(\mathcal{B}(D1), \mathcal{B}(D2), \mathcal{B}(Merge), \mathcal{B}(P2))$.

It is easy to verify the conditions for a simulation. The liveness properties of $S1$ and $S2$ follow trivially, since the single and double circles are identically distributed in the state diagrams.

5.1.5 verification of $T1$ and $T2$

Next, we verify that the composition of Si and $Plus1$ satisfy the specification of Ti . The procedure is analogous to the preceding case. In Fig. 5.1.7 are buffered trace generators that specify $T1$ and $T2$. There are only two event names, i and l , where i is input buffered.

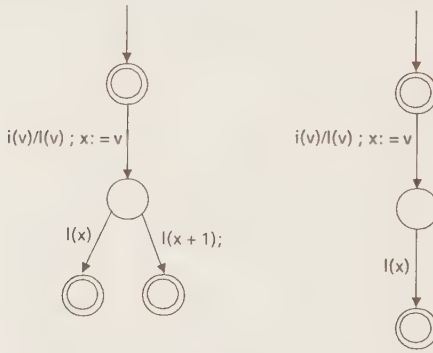


Fig. 5.1.7. The buffered trace generators $\mathcal{B}(T1)$ and $\mathcal{B}(T2)$.

To verify that the network satisfies the specification given by $\mathcal{B}(Ti)$, we must prove the implication $\Pi(\mathcal{B}(Si), \mathcal{B}(Plus1)) \supset \mathcal{B}(Ti)$ for $i = 1, 2$. When forming the product $\Pi(\mathcal{B}(Si), \mathcal{B}(Plus1))$, the buffer buf_l must be expanded, and the buffer buf_m can be removed according to theorem 4.4.12. The result is shown in Fig. 5.1.8. The state will now also contain the contents of the variable buf_l .

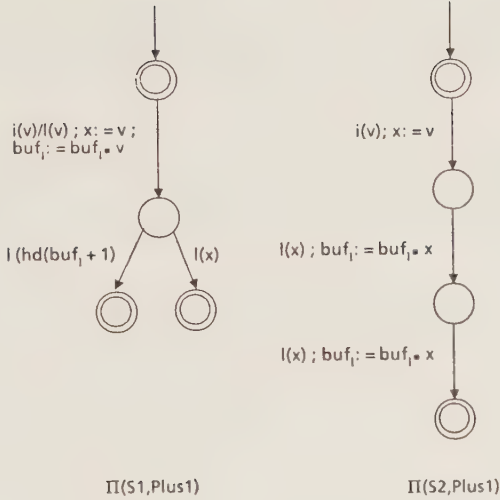


Fig. 5.1.8. $\Pi(\mathcal{B}(Si), \mathcal{B}(Plus1))$.

The last step is to show a simulation between $\Pi(\mathcal{B}(Si), \mathcal{B}(Plus1))$ and $\mathcal{B}(Ti)$. This follows similarly as in the preceding verification.

5.2 The Sliding Window Protocol of HDLC

In this section, we specify and verify a more complex example than in section 5.1: a protocol that ensures correct transfer of messages across a faulty medium. The protocol is a sliding window protocol of the same type as in the HDLC protocol.

The HDLC protocol is intended to provide reliable full-duplex data transfer between protocol entities, using an error-prone physical communication channel. The protocol includes procedures both for connection management and data transfer.

In this section, we shall specify and verify an idealized version of the data transfer procedures in HDLC ([ISO 79]). For instance, we only consider messages that transmit data and messages that transmit acknowledgments, we only consider data transfer in one direction. On the other hand, we prove liveness of the protocol by imposing certain fairness conditions on the communication channels. Thus our verification does not prove that HDLC as defined in the standard functions properly, but rather that the used procedures for message transfer rest on sound principles.

5.2.1 Description of the protocol

In this section, we describe the protocol that is verified in this section. The protocol consists of two entities, P_1 and P_2 , between which there are two media M_{12} and M_{21} , one in each direction, structured as in Fig. 5.2.1.

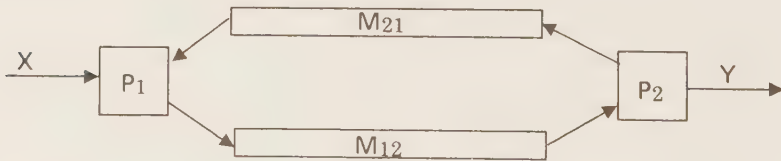


Fig. 5.2.1. Structure of the protocol

The purpose of the protocol is to transfer messages received on the input channel X to the channel Y in the same order as they were received on the channel X . The media M_{12} and M_{21} deliver messages after a finite but unbounded delay. The problem is that the media are not perfect: they may lose and corrupt messages, but not reorder them. We assume that the media contain a mechanism for detecting and discarding corrupted messages, so that corruption can be modeled as loss. Therefore P_1 and P_2 must employ a protocol with acknowledgments and retransmissions of messages.

The entities P_1 and P_2 employ a so-called sliding window protocol, in which messages that are sent over M_{12} are provided with sequence numbers. The medium M_{21} is used to send acknowledgments with sequence numbers in the other direction. The sequence numbers range from 0 to 7.

The entity P_1 provides each message received on X with a sequence number. The assignment of sequence numbers is cyclic, going from 0 to 7, and then starting at 0 again. Initially, P_1 transmits messages over M_{12} with consecutive sequence numbers 0, 1, 2, etc. Since the medium M_{12} is faulty, P_1 cannot know whether the messages will reach P_2 . Therefore P_1 also waits for incoming acknowledgments over M_{21} . The arrival of an acknowledgment with the sequence number n signals that P_2 has correctly received messages up to sequence number $n - 1$. The messages that have been sent by P_1 but not yet been acknowledged are called *outstanding messages*. There must never be more than 7 outstanding messages. Thus, after sending messages with sequence numbers 0 through 6, P_1 must wait for acknowledgments. If in this situation an acknowledgment with sequence number 3, say, arrives, then the messages with sequence numbers 0, 1, 2 are no longer outstanding, and P_1 may continue to send messages with sequence numbers 7, 0, and 1. If no acknowledgment arrives for any outstanding message, then presumably some of the messages have been lost in the medium. Therefore P_1 should resend outstanding messages after some period of time if no acknowledgment for them arrives.

The entity P_2 receives messages from M_{12} and uses their sequence numbers to check that they arrive in the correct order. A message which arrives in sequence, i.e., with a sequence number following that of the last correctly received message, is delivered to the channel Y . A message which arrives out of sequence is discarded. The entity P_2 must also send acknowledgments over M_{21} to signal what is the sequence number of the next expected message in the sequence.

The restriction that not more than 7 messages may be outstanding at P_1 is necessary to avoid confusion over the sequence numbers. For instance, if 8 outstanding messages were allowed, then P_1 could send messages with sequence numbers 0 through 7, and thereafter receive an acknowledgment with number 0. It would then be impossible for P_1 to tell whether P_2 has received no message or whether P_2 has received all messages. In the first case, P_1 should resend the messages, and in the second case P_1 should continue sending the next message.

To verify the protocol, we regard the entities and the medium as separate components which are connected according to Fig. 5.2.2.

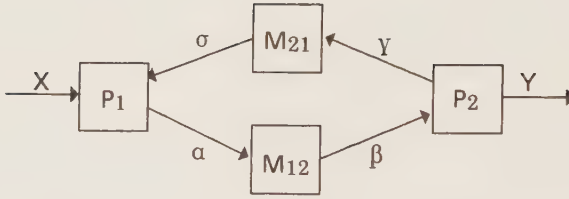


Fig. 5.2.2. Structure of the components of the protocol.

We shall specify each of the components, and verify that their composition transfers messages correctly from X to Y .

The events of the system are the following:

$X(d)$ and $Y(d)$ denote the transmission of the message d over channel X or Y , respectively.

$\alpha(d, n)$ and $\beta(d, n)$ denote the transmission of message d with sequence number n from P_1 to M_{12} and from M_{12} to P_2 , respectively.

$\gamma(n)$ and $\delta(n)$ denote the transmission of an acknowledgment with sequence number n from P_2 to M_{21} and from M_{21} to P_1 , respectively.

In the specifications, we use variables that range over messages, and variables that range over sequence numbers. Addition and subtraction will always be performed modulo 8.

The inequality $n_1 \leq n_2 \leq n_3$ means that $n_3 - n_1 \pmod{8} \geq n_2 - n_1 \pmod{8}$,

The inequality $n_1 \leq n_2 < n_3$ means that $n_3 - n_1 \pmod{8} > n_2 - n_1 \pmod{8}$.

5.2.2 Specification of Data Transfer

We first specify the desired behavior of the protocol: correct transfer of messages from channel X to channel Y . This specification is analogous to the buffer specification in example 4.3.5

Events:	$X(d)$ input buffered, $Y(d)$
Transitions:	$X(d) / Y(d)$
Liveness:	X : buffer guarantee

Fig. 5.2.3. Trace generator $\mathcal{B}(S)$.

The trace generator $\mathcal{B}(S)$ specifies a buffer, which transfers messages from the channel X to the channel Y and preserves their order. The buffer guarantee requirement implies that all messages in the buffer will eventually be delivered on Y .

5.2.3 Specification of the Components

Next we give specifications for each of the four components in Fig. 5.2.2. First we specify the entity P_1 .

Events:	$X(d), \delta(n)$, input buffered
	$\alpha(d, n)$
Variables:	b_1, b_2, w
Initialization:	$b_1 = b_2 = 0$
Transitions:	$X(d) : b_2 \neq b_1 + 7 \longrightarrow w[b_2] := d, b_2 := b_2 + 1 \quad (P_11)$ $\alpha(d, n) : b_1 \leq n < b_2 \wedge w[n] = d \longrightarrow NOP \quad (P_12)$ $\delta(n) : true \longrightarrow b_1 := n \quad (P_13)$
Liveness:	δ : buffer guarantee X : buffer fairness for each $N \in [0, \dots, 7]$: $\langle N = b_1 \neq b_2, \alpha(d, N) \rangle$

Fig. 5.2.4. Trace generator $\mathcal{B}(P_1)$.

We use NOP to denote that no state variable is changed in a transition. The expression $\alpha(d, N)$ in the fairness requirement is a shorthand for $\alpha \wedge (\exists d)[v_\alpha = N]$, which states that a message with sequence number N is transmitted over α . The sender communicates via events of form $X(d)$, $\alpha(d, n)$, and $\delta(n)$. The events $X(d)$ and $\delta(d, n)$ are input buffered. The outstanding messages are recorded in the variable w , which is an array with indices 0 through 7. An outstanding message with sequence number n is recorded as $w[n]$. The variables b_1 and b_2 record the sequences numbers of the outstanding messages, namely sequence numbers n such that $b_1 \leq n < b_2$. Initially there are no outstanding messages, so $b_1 = b_2 = 0$.

Transition (P_11) states that the next message from X can be made outstanding if there are less than 7 outstanding messages. The message from X is recorded in the array w , and b_2 is increased to record an extra outstanding message.

Transition (P_12) states that any message which is outstanding may be transmitted over α together with its sequence number. Transition P_13 describes how an acknowledgment for an outstanding message received on δ changes the index b_1 to record that messages are no longer outstanding.

The buffer guarantee requirement for $\delta(n)$ implies that all acknowledgments received over δ will eventually be processed. The buffer fairness requirement for $X(d)$ implies that if there are less than 7 outstanding messages, then eventually a message will be taken from the input buffer corresponding to channel X . It must also be ensured that messages are transmitted over α until they are acknowledged. Therefore there is a fairness requirement which states that, if infinitely often the first outstanding message has sequence number N , then infinitely often a message with sequence number N will be transmitted over α .

Note that we allow any outstanding message to be transmitted at any time.

The liveness requirements only require that the first outstanding message be re-transmitted until acknowledgment arrives. These requirements are more general than those in HDLC, where it is specified that the entire sequence of outstanding messages be resent if some time has elapsed without arrival of acknowledgment.

The entity P_2 is specified by the following trace generator:

Events:	$\beta(d, n)$ input buffered $\delta(n), Y(d)$
Variables:	$Next$
Initialization:	$Next = 0$
Transitions:	$\beta(d, n) / Y(d) : n = Next \rightarrow Next := Next + 1 \quad (P_{21})$ $\beta(d, n) : n \neq Next \rightarrow NOP \quad (P_{22})$ $\gamma(Next) : true \rightarrow NOP \quad (P_{23})$
Liveness:	β : buffer guarantee $true \hookrightarrow \gamma$

Fig. 5.2.5. Trace generator $\mathcal{B}(P_2)$.

In the specification of P_2 , the variable $Next$ records the sequence number of the next expected message which arrives in order. If a message with the expected sequence number appears, as in transition (P_{21}) , the message is delivered to Y . If a message with another sequence number appears, as in transition (P_{22}) , it is simply ignored. Acknowledgements that contain the sequence number of the next expected message are sent over γ (transition (P_{23})).

The buffer guarantee requirement means that all messages that arrive on β will eventually be considered. The fairness requirement implies that acknowledgments will be sent continuously.

The specifications of the media are rather similar to a specification of a perfect FIFO buffer, as in example 4.3.5. To specify that they may lose message, we introduce an extra transition with an event on the input channel, but no event on the output channel. The effect is that the message that corresponds to the input event is discarded.

Events:	$\alpha(d, n)$ input buffered $\beta(d, n)$
Transitions:	$\alpha(d, n) / \beta(d, n) \quad (M_{121})$ $\alpha(d, n) \quad (M_{122})$
Liveness:	α : buffer guarantee For each sequence number N : $\langle \alpha(d, N), \beta(d, N) \rangle$

Fig. 5.2.6. Trace generator $\mathcal{B}(M_{12})$.

Transition (M_{121}) delivers a message correctly, whereas transition (M_{122}) loses a message. The protocol will not work if the media continuously lose messages. If sufficiently many messages are transmitted to a medium, then some must be delivered. This is stated by the fairness requirement, which states that if infinitely many messages with a certain sequence number N are received, then infinitely many messages with sequence number N will be delivered.

M_{21} is specified analogously:

Events:	$\gamma(n)$ input buffered $\delta(n)$	
Transitions:	$\gamma(n) / \delta(n)$	(M_{211}) (M_{212})
Liveness:	γ buffer guarantee For each sequences number N :	$\langle \gamma(N), \delta(N) \rangle$

Fig. 5.2.7. Trace generator $\mathcal{B}(M_{21})$.

It should be noted that the protocol will not work under the weaker fairness requirement that a medium which receives infinitely many messages also deliver infinitely many messages. It is necessary to make such a requirement separately for each sequence number.

5.2.4 Verification of the protocol

We shall now prove that the network consisting of the four components P_1 , P_2 , M_{12} and M_{21} transfer data correctly. For this we establish the implication

$$\llbracket \Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21})) \rrbracket \supset \llbracket \mathcal{B}(S) \rrbracket$$

To prove this implication, we must first form the product of the buffered trace generators $\mathcal{B}(P_1)$, $\mathcal{B}(P_2)$, $\mathcal{B}(M_{12})$ and $\mathcal{B}(M_{21})$, and then prove that the product implies the trace generator $\mathcal{B}(S)$. Since the trace generators involved are buffered, we shall apply theorem 4.6.12. In example 4.6.14, we considered this example, and concluded that the buffers buf_α , buf_β , and buf_δ can be removed. The requirement of that these buffers are live is easily verified from the buffer guarantee requirements. Before forming the product, we must also expand the buffer corresponding to γ , since it occurs in both P_2 and M_{21} . The product then becomes the following buffered trace generator.

We must next establish a simulation between $\Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21}))$ and $\mathcal{B}(S)$. Since both buffered trace generators have the event name X as input buffered, we shall apply theorem 4.6.13. In other words, we first remove buf_X from $\Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21}))$, and prove that it implies the trace generator obtained by expanding the buffer of S . The precondition for this method, according to theorem 4.6.13 is that buf_X is live. Since there is a buffer fairness condition for X , we use proposition 4.6.9, which states that buf_X is live if infinitely often a transition that removes an element from buf_X is enabled, i.e., if it can be established that

$$(true) \xrightarrow[R]{\mathcal{F}} \Diamond (false ; b_2 \neq b_1 + 7)$$

where R is the simulation relation in the following proof. The proof of this is deferred until after the establishment of the simulation.

Events:	$X(d)$ input buffered $Y(d), \alpha(d, n), \beta(d, n), \gamma(n), \delta(n)$	
Variables:	$b_1, b_2, Next, w$ buf_γ	
Initialization:	$b_1 = b_2 = Next = 0$ $buf_\gamma = \langle \rangle$	
Transitions:	$X(d) : b_2 \neq b_1 + 7 \rightarrow w[b_2] := d, b_2 := b_2 + 1$	(T1)
	$\alpha(d, n) : (b_1 \leq n < b_2) \wedge w[n] = d \rightarrow NOP$	(T2)
	$\alpha(d, n) / \beta(d, n) / Y(d) :$ $(b_1 \leq n < b_2) \wedge (w[n] = d) \wedge (n = Next) \rightarrow Next := Next + 1$	(T3)
	$\alpha(d, n) / \beta(d, n) :$ $(b_1 \leq n < b_2) \wedge (w[n] = d) \wedge (n \neq Next) \rightarrow NOP$	(T4)
	$\gamma(Next) : true \rightarrow buf_\gamma := buf_\gamma \bullet Next$	(T5)
	$\tau : buf_\gamma \neq \langle \rangle \rightarrow buf_\gamma := tl(buf_\gamma)$	(T6)
	$\delta(hd(buf_\gamma)) : buf_\gamma \neq \langle \rangle \rightarrow$ $buf_\gamma := tl(buf_\gamma)$ $b_1 := hd(buf_\gamma)$	(T7)
Liveness:	X : buffer fairness $buf_\gamma \neq \langle \rangle \hookrightarrow buf_\gamma^{new} = tl(buf_\gamma^{old})$ $true \hookrightarrow \gamma$ for each $N \in [0 \dots 7]$: $\langle N = b_1 \neq b_2, \alpha(d, N) \rangle$ $\langle \alpha(d, N), \beta(d, N) \rangle$ $\langle \gamma(N), \delta(N) \rangle$	

Fig. 5.2.8. $\Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21}))$.

After expanding the buffer, $\mathcal{B}(S)$ has the form below:

Events:	$X(d), Y(d)$	
Variables:	buf_X	
Initialization:	$buf_X = \langle \rangle$	
Transitions:	$X(d) : true \rightarrow buf_X := buf_X \bullet d$	(S1)
	$Y(d) : buf_X \neq \langle \rangle \rightarrow buf_X := tl(buf_X)$	(S2)
Liveness:	$buf_X \neq \langle \rangle \hookrightarrow buf_X^{new} := tl(buf_X^{old})$	

Fig. 5.2.9. Expanding the buffer in $\mathcal{B}(S)$.

The simulation relation R consists of the following conjuncts.

$$b_1 \leq Next \leq b_2 \quad (R1)$$

$$buf_\gamma \in b_1^*(b_1 + 1)^* \dots Next^* \quad (R2)$$

$$buf_X = \left\{ \begin{array}{l} \text{if } Next = b_2 \text{ then } \langle \rangle \\ \text{else } w[Next] \bullet w[Next + 1] \bullet \dots \bullet w[b_2 - 1] \end{array} \right\} \quad (R3)$$

The conjunct (R1) relates the sequence numbers of the outstanding messages at P_1 with the next sequence number expected by the receiver. The sequence number expected by the receiver is a number of an outstanding message, or the sequence number immediately following the highest number of an outstanding message. (R2) states that the buffer buf_γ contains sequence numbers which range from the lowest number of an outstanding message to the number expected by P_2 . (R3) relates buf_X to the state of $\Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21}))$. The

buffer buf_X contains the sequence of messages that have been received at X , but not delivered at Y .

We must verify that R , as given by $R1$, $R2$, and $R3$ indeed satisfies the conditions for being a simulation. It is trivial to verify that the initial conditions satisfy R . To verify that R is indeed a simulation, we shall prove that

- $T1$ is simulated by $S1$
- $T3$ is simulated by $S2$.
- $T2$, $T4$, $T5$, $T6$, and $T7$ are simulated by stuttering transitions of S .

We consider each transition of the product separately.

$T1$ is simulated by $S1$. The variables of $R2$ are not affected by these transitions. To verify that $R1$ and $R3$ are preserved, we need to verify the following two implications

$$\begin{aligned} [(b_2 \neq b_1 + 7) \wedge (b_1 \leq Next \leq b_2)] \supset b_1 \leq Next \leq b_2 + 1 \\ buf_X = w[Next] \bullet \dots \bullet w[b_2 - 1] \supset buf_X \bullet d = w[Next] \bullet \dots \bullet w[b_2 - 1] \bullet d \end{aligned}$$

$T2$ does not change any variables of $\Pi(\mathcal{B}(P_1), \mathcal{B}(P_2), \mathcal{B}(M_{12}), \mathcal{B}(M_{21}))$.

$T3$ is simulated by $S2$. To verify that $R1$, $R2$, and $R3$ are preserved, we need to verify the following implication

$$\left\{ \begin{array}{l} [(n = Next) \wedge (b_1 \leq n < b_2)] \\ buf_\gamma \in (b_1)^*(b_1 + 1)^* \dots (Next)^* \\ buf_X = w[Next] \bullet \dots \bullet w[b_2 - 1] \end{array} \right\} \supset \left\{ \begin{array}{l} b_1 \leq Next + 1 \leq b_2 \\ buf_\gamma \in (b_1)^*(b_1 + 1)^* \dots (Next + 1)^* \\ tl(buf_X) = w[Next + 1] \bullet \dots \bullet w[b_2 - 1] \end{array} \right\}$$

$T4$ does not change any variables.

$T5$ adds one element to buf_γ . We must verify that $R2$ is preserved, as follows:

$$buf_\gamma \in (b_1)^*(b_1 + 1)^* \dots (Next)^* \supset buf_\gamma \in (b_1)^*(b_1 + 1)^* \dots (Next)^* Next$$

$T6$ deletes an element from buf_γ . We must verify that

$$\left\{ \begin{array}{l} buf_\gamma \neq \langle \rangle \wedge \\ buf_\gamma \in (b_1)^*(b_1 + 1)^* \dots (Next)^* \end{array} \right\} \supset \left\{ tl(buf_\gamma) \in (b_1)^*(b_1 + 1)^* \dots (Next)^* \right\}$$

$T7$ deletes an element from buf_γ , and updates b_1 . We must verify that

$$\left\{ \begin{array}{l} buf_\gamma \neq \langle \rangle \wedge \\ b_1 \leq Next \leq b_2 \wedge \\ buf_\gamma \in (b_1)^* \dots (Next)^* \end{array} \right\} \supset \{ tl(buf_\gamma) \in hd(buf_\gamma)^* (hd(buf_\gamma) + 1)^* \dots (Next)^* \}$$

In the second part of the proof, we must establish the liveness requirement of $\mathcal{B}(S)$, and that the buffer corresponding to X is live.

$$(1) (buf_X \neq \langle \rangle) \xrightarrow[R]{\mathcal{F}} \Diamond (buf_X^{new} := tl(buf_X^{old}); false)$$

$$(2) (true) \xrightarrow[R]{\mathcal{F}} \Diamond (false; b_2 \neq b_1 + 7)$$

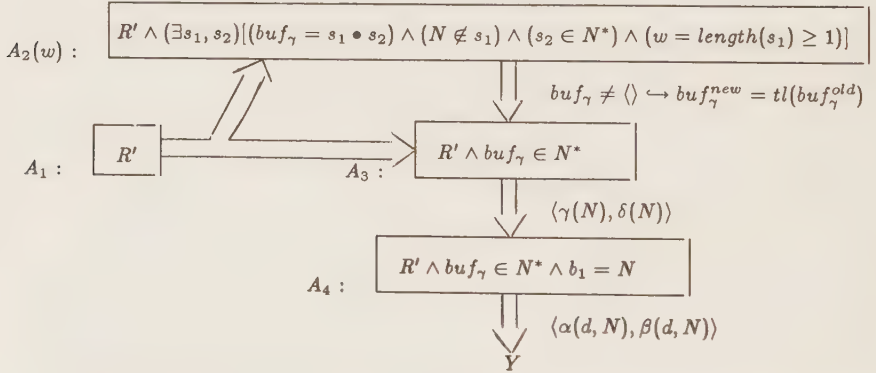
where $\mathcal{F}$ is the liveness requirements of the product. Requirement (1) is (by the consequence rule) equivalent to the formula

$$(Next \neq b_2) \xrightarrow[R]{\mathcal{F}} \Diamond (false; Y).$$

We prove this formula by an application of the chain rule. Let R' denote the formula

$$R \wedge buf_X \neq \langle \rangle \wedge Next = N \neq b_2$$

The proof is illustrated in the following proof diagram.



The assertions in the diagram are labeled by $A_1, \dots, A_4$. The individual steps of the diagram are motivated as follows:

A_1 It holds that $A_1 \supset ((\exists w)[A_2(w)] \vee A_3)$.

$A_2(w)$ states that buf_γ contain $w \geq 1$ sequence numbers that are different from N followed by a sequence of N 's. The diagram states that eventually w will decrease or A_3 will become true, unless Y occurs. This follows by an application of the guarantee rule using the guarantee requirement $buf_\gamma \neq \langle \rangle \hookrightarrow buf_\gamma^{new} = tl(buf_\gamma^{old})$, which states that an element will be removed from buf_γ unless buf_γ is empty. The premises of the guarantee rule are

- (i) If no event of form Y occurs, then w is not increased by any transition. The reason is that $Next = N$ holds unchanged, and only the element $Next$ can be inserted into buf_γ .

(ii) When an element is removed from buf_γ , the number of non- N elements in buf_γ decreases. Thus w decreases, or A_3 becomes true.

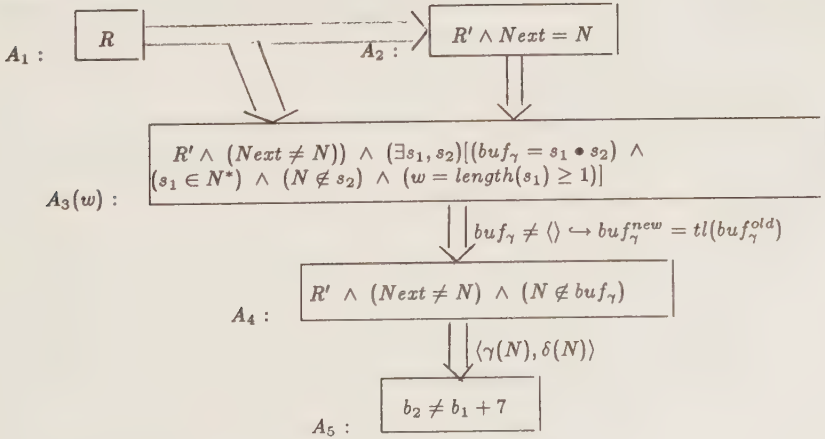
(iii) $w \geq 1$ trivially implies that buf_γ is non-empty.

A_3 states that buf_γ contains only elements N . The diagram states that unless Y occurs, b_1 will become equal to N . This follows from an application of the fairness rule, using the fairness requirement $\langle \gamma(N), \delta(N) \rangle$, since an occurrence of $\delta(N)$ causes $b_1 := N$. We must show the premise $(A_3) \xrightarrow[R]{\mathcal{F}} \Diamond (\gamma(N) ; \text{false})$. This follows trivially from the guarantee requirement $\text{true} \hookrightarrow \gamma$

A_4 states that buf_γ contains only elements N and that $b_1 = N$. To verify that $Y(d)$ must eventually occur, we apply the fairness rule using the fairness requirement $\langle \alpha(d, N), \beta(d, N) \rangle$. The premises are the following:

- (i) Unless $Y(d)$ occurs, A_4 remains true, which is easy to verify.
- (ii) When $\beta(d, N)$ occurs, $Y(d)$ also occurs, according to transition (T3).
- (iii) $(A_4) \xrightarrow[R]{\mathcal{F}} \Diamond (\alpha(d, N) ; \text{false})$ is established by the fairness rule and the fairness requirement $\langle N = b_1 \neq b_2, \alpha(d, N) \rangle$ whose left formula is implied by A_4 , since $buf_X \neq \langle \rangle$ implies $b_1 \neq b_2$.

Next we verify the second liveness formula, stating that buf_X is live. Recall that we must prove $(\text{true}) \xrightarrow[R]{\mathcal{F}} \Diamond (\text{false} ; b_2 \neq b_1 + 7)$. Let R' be the assertion $R \wedge (b_1 = N) \wedge (b_2 = N + 7)$. The proof is illustrated in the following proof diagram.



The motivations for these steps are as follows:

R implies the disjunction of A_2 and A_5 .

A_2 implies the disjunction of A_3 and A_4 .

A_3 states that buf_γ contains $w \geq 1$ elements N followed by non- N elements.

The motivation for progress is similar to the motivation for A_2 in the preceding proof.

A_4 states that buf_γ contains only non- N elements. The motivation for progress is similar to the motivation for A_3 in the preceding liveness proof.

5.2.5 Discussion

HDLC and related protocols have been widely used to illustrate many specification and verification methods.

Shankar and Lam ([SL 83]) specify a full version of HDLC, and verify safety properties of the protocol by establishing a number of invariants. They do not consider liveness properties. Instead, they consider the real-time behavior of the protocol by including time-variables that represent elapsed time. Instead of verifying that “eventually, each message will be transferred,” they consider specifications of the form “if within a time duration T a message is not transferred, then the at least n retransmissions of the message have occurred, all of which have failed.”

Higashino et al ([HMSTK 84]) uses algebraic specification methods to specify the information transfer procedures of HDLC. The state of an entity is modeled as an element in an algebra, on which operations that correspond to transitions are defined. Within the algebraic framework, invariants can be formulated and proven, verifying safety properties of the protocol.

Brand and Joyner ([BJ 82]) verify HDLC by symbolic execution, using an automated verification system. With certain restrictions, they prove safety properties of the protocol. To a large extent the verification was carried out automatically.

None of the above approaches considers liveness properties of the protocol. For the related alternating-bit protocol, which is similar to the protocol verified here but with a sequence number ranging over only 0 and 1, liveness properties have been considered by other verification methods. Here we discuss the approaches most related to ours.

Lamport ([La 83]) verifies the alternating-bit protocol using the approach with state functions. The state functions play a role related to our state variables. To verify that a collection of components conforms to a specification, one must find a mapping from the state functions that specify the components to the state functions that specify the whole system. This can be regarded as a

special case of our simulation technique.

Lamport uses temporal logic to specify and verify liveness properties. He presents a proof of liveness, whose structure is related to our proof of liveness of HDLC.

Stark ([St 84]) specifies and verifies the alternating-bit protocol using transition systems with liveness requirements formulated in temporal logic. Safety properties are proven by a technique similar to our simulation technique. Liveness properties are formulated and proven in temporal logic. The liveness properties are formulated in a different style than ours and Lamport's, and have the form of rely-guarantee conditions ([St 85]). For example, for the entity P_1 of the protocol, Stark gives a liveness property of the form "If the entity can rely on the fact that sufficiently many transmissions of a message will eventually result in the receipt of an acknowledgment for the message, then P_1 will eventually process each message given to it from the environment." In some cases, this style of liveness properties lead to concise proofs of liveness properties of a system. Since they are sometimes rather complex, it seems that they are more difficult to verify from other specifications than the liveness requirements of ours and Lamport's approach.

5.3 An Algorithm for Concurrency Control in Multiple Copy Databases

In this section, we specify and verify an algorithm for maintaining consistency in a distributed database. The scenario is the following. Consider a database, which is distributed over n sites. At each site the database is represented by a *database manager* (DBM), as shown in Fig. 5.3.1.

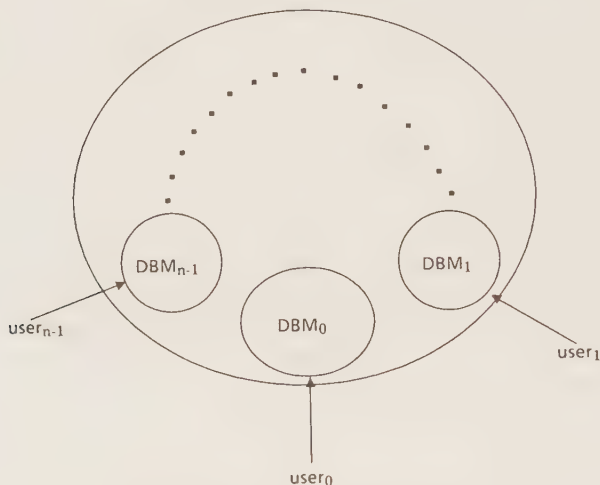


Fig. 5.3.1. A distributed database.

The data in the database is replicated so that each site contains a copy of each datum in the database. The database can be accessed by users at any of its sites. The users submit requests that either ask for the current value of a datum, or request that a datum be updated. The problem is to ensure that the database is consistent, i.e., that the different replicas of the database are the same, except for possible delays in propagation of changes. Since several different requests for updates to a datum can be issued at different sites, some procedure must be devised for deciding which update requests to honor. Assume for instance that two requests for updates of the datum x are made at different sites, one for $x := x + 1$, and one for $x := x + 2$. At most one of the requests can be carried out, and the other must be rejected, otherwise the consistency of the database would be destroyed.

One algorithm for ensuring consistency of a distributed database is due to Thomas ([Th 79]). The algorithm uses a voting scheme for deciding which update requests to honor. When an update request is received by a database manager, the request is passed around among managers. Each manager votes whether to accept the request or whether to reject it. An update request is honored only

if a majority of the managers accept it, otherwise it is rejected. This procedure ensures consistency, because for any two consecutive update requests that are honored, there must be at least one manager that accepts both of them, thus ensuring that they do not conflict.

5.3.1 Informal Description of the Algorithm

We shall specify each database manager, and verify that the composition of the managers, i.e., the database, can be regarded as one global database where updates do not conflict.

For simplicity, we assume that the database contains only one data element x , and that each manager has a local copy of x . At any moment, a user can ask a manager for the value of its local copy of x , or request that the value of x be updated.

To distinguish between different updates, to detect requests for updates to obsolete values, and to assign priorities to requests, each value of x has an associated time-stamp. The time-stamp of a value v contains information about the time at which the request for an update to v was issued by a user, and the site at which the request was issued. We use $origin(v)$ to denote the site at which the request for update to v was issued. The values of time-stamps are ordered. Thus $v_1 < v_2$ if the time of v_1 is earlier than the time of v_2 , or the times are the same and $origin(v_1)$ has a lower index than $origin(v_2)$. The time-stamps are also used to assign priorities to update requests. For two update requests, one from v to v' , and the other from u to u' , we say that the first has priority over the second, denoted $(u, u') <_{prio} (v, v')$, if $u < v$, or if $u = v$ and $u' < v'$.

Assume that a user issues a request to update the value of the datum from v to v' . The database manager then constructs a request message containing the values v and v' , and slots for inserting votes. The request message is passed around the database managers. We assume that the message is passed circularly among the managers. Each manager adds his vote to the request message and then passes it to the next manager. The rules for voting are the following for a manager whose local copy of the datum has the value x_i .

- (V1) Vote OK if $x_i \leq v$ (i.e., the updated value is not obsolete), and the manager has not voted OK for any pending request.
- (V2) Vote PASS if $x_i \leq v$ and the manager has voted OK for an update request with higher priority.
- (V3) Defer voting and keep the request message if $x_i \leq v$ and the manager has voted OK for a request to an update with lower priority.

The request message is passed around until a manager has enough information to decide whether to reject or accept the request. In this case, the manager

stops passing the request message around, and instead broadcasts a resolution message, telling the other managers whether the update requests was rejected or accepted. The rules for this are:

- (U1) If the number of OK votes constitutes at least half of the managers, the request is accepted.
- (U2) If the number of PASS votes constitutes a majority of the managers, the request is rejected since it will be impossible to get a majority of OK votes.
- (U3) If the updated value v of request is older than the current value at the manager, i.e., $v < x_i$, then the request is rejected, since it updates an obsolete value.

When a manager receives a resolution message for an update to v' , the manager updates its copy x_i of the datum if $x_i < v'$.

5.3.2 Specification of the components.

We shall specify each database manager DBM_i by a stepwise specification $\mathcal{A}(DBM_i)$. The managers communicate with each other, and with the users, by the following communication events.

$Req_i(v, v', o, p)$ is a request message sent by DBM_i to DBM_{i+1} (addition modulo n). The request message is a request to update the value from v to v' , for which so far o managers have voted OK, and p managers have voted PASS.

$Res_k(v, v', r)$ for $k = 0, \dots, n-1$ is a resolution message sent by DBM_k to all other managers, stating that the request to update from v to v' has been rejected (in case $r = rej$) or accepted (in case $r = acc$).

$Query_i$ is a request from a user at site i for the value of x

$Answer_i(v)$ tells the user at i that the current local value of the datum is v .

$Update_i(v, v')$ is a request to update the datum from v to v' .

$Notify_i(v, v', r)$ tells the user at i that the request to update the datum from v to v' was decided as r , where r is either rej or acc . Here i is the site at which the request was issued.

The variables of the stepwise specification $\mathcal{A}(DBM_i)$ are the following:

x_i is the local copy of the datum x .

q_i is true if a user at site i has requested the current value of x_i , and not received an answer message.

$stamp_i$ is the time-stamp of the last update request received. This variable is used to ensure that timestamps of subsequent update requests increase strictly.

$voted_i$ is true if DBM_i has voted OK for an update request, but still does not know whether it is accepted or rejected.

$voteval_i$ is the update (v, v') for which DBM_i has voted if $voted_i$ is true.

rgb_i is the bag of request messages that DBM_i has received but not answered

rsb_i is the bag of resolution messages that DBM_i has received but not considered.

nfb_i is the bag of resolution messages that DBM_i has processed, for updates originating at site i , for which notifications about the outcome of the request must be sent to the user.

The Initial state of $A(DBM_i)$ satisfies

$$x_i = X \quad \text{and} \quad \neg q_i \quad \text{and} \quad \neg voted_i \quad \text{and} \quad rgb_i = rsb_i = nfb_i = \emptyset$$

The Transitions of $A(DBM_i)$ are shown in Fig. 5.3.2. In these transitions we use extra auxiliary variables: v and v' range over values of the data item, r ranges over the set $\{acc, rej\}$, and o and p range over nonnegative integers.

Transitions $(t1) - (t4)$ constrain the communication between the database manager and the user. When a query is received from the user, as in transition $(t1)$, the variable q_i becomes *true*, and it is then possible to send back an answer with the current value of x_i , as in transition $(t2)$. When a request is received to update the value from v to v' , as in transition $(t3)$, the database manager first checks the timestamps. The timestamp of v must not be more recent than that of x_i , since x_i is the most recent value that a user of site i can have obtained. The timestamp of the proposed value v' must be more recent than $stamp_i$ to ensure that subsequent update requests have increasing timestamps. If the timestamps are in order, a request message is constructed with so far 0 votes, and the record $stamp_i$ of the last timestamp is updated. Transition $(t4)$ states that a notification may be communicated to the user for a pending notification in nfb_i .

In transitions $(t5)$ and $(t6)$, DBM_i receives request and resolution messages from other managers, and puts them into the corresponding bags of incoming messages that will subsequently be processed.

The next transitions specify the voting rules. Transition $(t7)$ corresponds to voting rule (V1), and transition $(t8)$ corresponds to voting rule (V2). The resolution rule (U1) corresponds to transition $(t9)$, and rule (U2) corresponds to transition $(t10)$. In these transitions, an element is removed from the bag rgb_i of pending request messages, since that request message has now been considered by DBM_i .

Transition $(t11)$ gives the rule for handling resolution messages that have arrived in rsb_i . If the update is accepted ($r = acc$) and more recent than DBM_i 's current copy of the datum ($x_i < v'$), then DBM_i 's copy is updated. If DBM_i

$$Query_i : true \longrightarrow q_i = true \quad (t1)$$

$$Answer_i(x_i) : q_i \longrightarrow q_i := false \quad (t2)$$

$$Update_i(v, v') :$$

$$true \longrightarrow \text{if } \left\{ \begin{array}{l} i = origin(v') \wedge \\ v \leq x_i < v' \wedge stamp_i < v' \end{array} \right\} \text{ then } \left\{ \begin{array}{l} rqb_i := rqb_i \cup \{Req(v, v', 0, 0)\} \\ stamp_i := v' \end{array} \right\} \quad (t3)$$

$$Notify_i(v, v', r) : Notify_i(v, v', r) \in nfb_i \longrightarrow \{ nfb_i := nfb_i \setminus \{Notify_i(v, v', r)\} \} \quad (t4)$$

$$Req_{i-1}(v, v', o, p) : true \longrightarrow rqb_i := rqb_i \cup \{Req_{i-1}(v, v', o, p)\} \quad (t5)$$

$$Res_k(v, v', r) : true \longrightarrow rsb_i := rsb_i \cup \{Res(v, v', r)\} \quad \text{for } k \neq i \quad (t6)$$

$$Req_i(v, v', o, p) :$$

$$\left\{ \begin{array}{l} Req(v, v', o-1, p) \in rqb_i \wedge \\ \neg voted_i \wedge o \leq n/2 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} rqb_i := rqb_i \setminus \{Req(v, v', o-1, p)\} \\ voted_i := true, voteval_i := (v, v') \end{array} \right\} \quad (t7)$$

$$Req_i(v, v', o, p) :$$

$$\left\{ \begin{array}{l} Req(v, v', o, p-1) \in rqb_i \wedge \\ voted_i \wedge (v, v') <_{prio} voteval_i \wedge \\ p < n/2 \end{array} \right\} \longrightarrow \{ rqb_i := rqb_i \setminus \{Req(v, v', o, p-1)\} \} \quad (t8)$$

$$Res_i(v, v', acc) :$$

$$\left\{ \begin{array}{l} Req(v, v', o, p) \in rqb_i \wedge \\ \neg voted_i \wedge o+1 > n/2 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} rqb_i := rqb_i \setminus \{Req(v, v', o, p)\} \\ rsb_i := rsb_i \cup \{Res(v, v', acc)\} \end{array} \right\} \quad (t9)$$

$$Res_i(v, v', rej) :$$

$$\left\{ \begin{array}{l} Req(v, v', o, p) \in rqb_i \wedge \\ v < x_i \vee \left(\begin{array}{l} voted_i \wedge \\ (v, v') <_{prio} voteval_i \wedge \\ (p+1 \geq n/2) \end{array} \right) \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} rqb_i := rqb_i \setminus \{Req(v, v', o, p)\} \\ rsb_i := rsb_i \cup \{Res(v, v', rej)\} \end{array} \right\} \quad (t10)$$

$$\tau : Res(v, v', r) \in rsb_i \longrightarrow \left\{ \begin{array}{l} rsb_i := rsb_i \setminus \{Res(v, v', rej)\} \\ \text{if } r = acc \wedge v' > x_i \text{ then } x_i := v' \\ \text{if } (v, v') = voteval_i \text{ then } voted_i := false \\ \text{if } i = origin(v') \text{ then } nfb_i := nfb_i \cup \{Notify_i(v, v', r)\} \end{array} \right\} \quad (t11)$$

Fig. 5.3.2. Transitions of DBM_i .

has voted for that message, then the vote is released, so that DBM_i can vote again. Finally, if the request originated from the user at site i , then the result is put into nfb_i so that the user is eventually notified about the outcome.

The **Liveness Requirements** of DBM_i are all simple guarantee conditions that ensure that the manager will progress. The liveness requirements $(L_i2) - (L_i4)$ are parameterized by parameters v, v' , and r . Thus for each value of these parameters, there is one guarantee requirement. In transition formulas, we use the notation $Notify_i(v, v', r)$ to denote that the event $Notify_i$ occurs with the parameters v, v' , and r . The formula can be regarded as a shorthand for

$$event = Notify_i \wedge v_{Notify_i} = \langle v, v', r \rangle$$

The liveness requirement are then the following:

$$q_i \hookrightarrow Answer_i \quad (L_i1)$$

$$Notify(v, v', r) \in nfb_i \hookrightarrow Notify_i(v, v', r) \quad (L_i2)$$

$$Res(v, v', r) \in rsb_i \hookrightarrow rsb_i^{new} = rsb_i^{old} \setminus \{Res(v, v', r)\} \quad (L_i3)$$

$$\left\{ \begin{array}{l} Req(v, v', o, p) \in rqbi \wedge \\ (voted_i \supset ((v, v') <_{prio} voteval_i)) \end{array} \right\} \hookrightarrow \left\{ \begin{array}{l} rqbi^{new} = rqbi^{old} \setminus \{Req(v, v', o, p)\} \\ \vee voteval_i^{new} <_{prio} (v, v') \end{array} \right\} \quad (L_i4)$$

Requirement (L_i1) states that the manager will eventually answer each query for a value in the database. Requirement (L_i2) states that if a resolution message corresponding to a request originating at site i has been processed by DBM_i , then a notification message must eventually be issued to the user. Requirement (L_i3) states that each pending resolution message in rsb_i must eventually be processed. Requirement (L_i4) states that if there is a pending request message in $rqbi$, for which voting shall not be deferred according to rule (V3), then it must eventually be removed from $rqbi$ unless the value of $voteval_i$ changes so that voting shall be deferred.

5.3.3 Specification of the Database

We shall next specify the intended behavior of the entire database. In the ideal case, the database should behave as if it contained only one copy of the datum, to which queries and updates from all sites were made. However, this condition is too strong, since it takes time to propagate updates between the managers. The database therefore behaves as if it contained several values of the datum. Consistency is obtained by requiring that only the most recent of these values may be updated. Any update request to a value which is older than the most recent will be rejected.

$$q_i : \text{true} \longrightarrow q_i := \text{true} \quad (s1)$$

$$\text{Answer}_i(x_i) : q_i \longrightarrow q_i := \text{false} \quad (s2)$$

$$\text{Update}_i(v, v') :$$

$$\text{true} \longrightarrow \text{if} \left\{ \begin{array}{l} i = \text{origin}(v') \wedge \\ v \leq x_i < v' \wedge \text{stamp}_i < v' \end{array} \right\} \text{then} \left\{ \begin{array}{l} \text{rgb} := \text{rgb} \cup \{\text{Req}(v, v')\} \\ \text{stamp}_i := v' \end{array} \right\} \quad (s3)$$

$$\text{Notify}_i(v, v', r) : \left\{ \begin{array}{l} \text{Notify}(v, v', r) \in \text{nfb} \wedge \\ i = \text{origin}(v') \end{array} \right\} \longrightarrow \text{nfb} := \text{nfb} \setminus \{\text{Notify}(v, v', r)\} \quad (s4)$$

$$\tau : \left\{ \begin{array}{l} \text{Req}(v, v') \in \text{rgb} \wedge \\ v = \text{latest}(xseq) \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} xseq := xseq \cup \{v'\} \\ \text{rgb} := \text{rgb} \setminus \{\text{Req}(v, v')\} \\ \text{nfb} := \text{nfb} \cup \{\text{Notify}(v, v', \text{acc})\} \end{array} \right\} \quad (s5)$$

$$\tau : \text{Req}(v, v') \in \text{rgb} \longrightarrow \left\{ \begin{array}{l} \text{rgb} := \text{rgb} \setminus \{\text{Req}(v, v')\} \\ \text{nfb} := \text{nfb} \cup \{\text{Notify}(v, v', \text{rej})\} \end{array} \right\} \quad (s6)$$

$$\tau : x_i < v' \wedge v' \in xseq \longrightarrow x_i := v' \quad (s7)$$

Fig. 5.3.3. Transitions of *DB*.

The database is specified by the stepwise specification $\mathcal{A}(DB)$. The event types of $\mathcal{A}(DB)$ are the event types of the $\mathcal{A}(DBM_i)$ that communicate with the users, namely $Query_i$, $Answer_i(x)$, $Update_i(v, v')$, and $Notify_i(v, v', r)$ for all $i = 1, \dots, n$.

The **Variables** of $\mathcal{A}(DB)$ are x_i , q_i , and $stamp_i$ for $i = 1, \dots, n$, with the same functions as among the $\mathcal{A}(DBM_i)$'s, and the following variables:

$xset$ contains the values of all updates that have been accepted at some point in the past. The most recent of these, given by $latest(xset)$, represents the "current" value of the datum, although it may not yet have reached all database managers.

rgb contains all pending update requests of form $Req(v, v')$ whose fate have not yet been decided.

nfb contains pending notifications of form $Notify(v, v', r)$ for update requests whose fate have been decided but not reported to the originating user.

Initially, we assume that the value of the datum in the database is X . The **initial state** satisfies

$$Seq = \{X\}$$

$$x_1 = \dots = x_n = X$$

$$q_1 = \dots = q_n = false$$

$$rgb = nfb = \emptyset$$

The **transitions** are shown in Fig. 5.3.3. The first four transitions ($s1$) – ($s4$) have a similar function as the transitions ($t1$) – ($t4$) of DBM_i .

Transition ($s5$) corresponds to the requirement that an update request may be accepted only if it updates the most recent value in $xset$. If an update request is accepted, the new value is added to $xset$. According to transition ($s6$), an update request may also be rejected at any moment. In both transitions ($s5$) and ($s6$), the pending request is removed from rgb and a pending notification about the outcome accept/reject is put into nfb . Transition ($s7$) corresponds to the propagation of updated values in the database. A local value at a site may be replaced by a more recent value.

The **Liveness Requirements** of DB are the following, Requirement (L2) is parameterized by v, v' and r . Requirements (L1) and (L3) are parameterized by i .

$$q_i \hookrightarrow Answer_i \quad (L1)$$

$$Notify(v, v', r) \in nfb \hookrightarrow Notify_{origin(v')}(v, v', r) \quad (L2)$$

$$x_i < latest(xseq) \hookrightarrow x_i^{new} > x_i^{old} \quad (L3)$$

$$(\exists j, v, v') \left[\begin{array}{l} Req(v, v') \in rqb \wedge \\ v \geq latest(xset) \end{array} \right] \hookrightarrow \left[\begin{array}{l} [latest(xset^{new}) > latest(xset^{old})] \\ \vee (\exists i) Update_i \end{array} \right] \quad (L4)$$

Requirement (L1) states that queries will eventually be answered. Requirement (L2) states that if the fate of a request has been decided, then the user that issued the request will eventually be notified. Requirement (L3) states that a local copy of the datum which is not most current, will eventually be replaced by a more recent value.

The requirements (L1) – (L3) do not state that decisions will be taken on requests for updates. A desirable liveness condition is that each pending update request will be either rejected or accepted, and that if some update requests are pending, then at least one of them must be accepted. However, this strong condition is not satisfied by the algorithm. If update requests arrive sufficiently often to the database, then votes may be distributed among too many requests so that a majority is never reached. We therefore state the weaker liveness requirement (L4), which states that if update requests for values which are not obsolete are pending, and no further update requests arrive, then a request must eventually be accepted.

5.3.4 Verification of the Algorithm

To verify that the algorithm specified for each database manager satisfies the specification $\mathcal{A}(DB)$, we must establish the implication

$$\llbracket \Pi(\mathcal{A}(DBM_0), \dots, \mathcal{A}(DBM_{n-1})) \rrbracket \supseteq \llbracket \mathcal{A}(DB) \rrbracket$$

The product $\Pi(\mathcal{A}(DBM_0), \dots, \mathcal{A}(DBM_{n-1}))$ of the specifications of the managers contains the variables $x_i, q_i, stamp_i, voted_i, voteval_i, rqb_i, rsb_i$, and nfb_i for $i = 0, \dots, n-1$. The initial state satisfies the conjunction of the initial states of $DBM_0, \dots, DBM_{n-1}$. The transitions of the product are shown in Fig. 5.3.4. They are very similar to the transitions of the individual $\mathcal{A}(DBM_i)$'s. Transitions (t5) and (t6) now occur coupled with other transitions among (t7) – (t10). The liveness requirements are the union of the liveness requirements of each $\mathcal{A}(DBM_i)$.

$$\text{Query}_i : \text{true} \longrightarrow q_i = \text{true} \quad (T1)$$

$$\text{Answer}_i(x_i) : q_i \longrightarrow q_i := \text{false} \quad (T2)$$

$$\text{Update}_i(v, v') :$$

$$\text{true} \longrightarrow \text{if} \left\{ \begin{array}{l} i = \text{origin}(v') \wedge \\ v \leq x_i < v' \wedge \text{stamp}_i < v' \end{array} \right\} \text{then} \left\{ \begin{array}{l} \text{rgb}_i := \text{rgb}_i \cup \{\text{Req}(v, v', 0, 0)\} \\ \text{stamp}_i := v' \end{array} \right\} \quad (T3)$$

$$\text{Notify}_i(v, v', r) : \text{Notify}(v, v', r) \in \text{nfb}_i \longrightarrow \left\{ \text{nfb}_i := \text{nfb}_i \setminus \{\text{Notify}(v, v', r)\} \right\} \quad (T4)$$

$$\text{Req}_i(v, v', o, p) :$$

$$\left\{ \begin{array}{l} \text{Req}(v, v', o-1, p) \in \text{rgb}_i \wedge \\ \neg \text{voted}_i \wedge o \leq n/2 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} \text{rgb}_i := \text{rgb}_i \setminus \{\text{Req}(v, v', o-1, p)\} \\ \text{rgb}_{i+1} := \text{rgb}_{i+1} \cup \{\text{Req}(v, v', o, p)\} \\ \text{voted}_i := \text{true}, \text{voteval}_i := (v, v') \end{array} \right\} \quad (T5)$$

$$\text{Req}_i(v, v', o, p) :$$

$$\left\{ \begin{array}{l} \text{Req}(v, v', o, p-1) \in \text{rgb}_i \wedge \\ \text{voted}_i \wedge (v, v') <_{\text{prio}} \text{voteval}_i \wedge \\ p < n/2 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} \text{rgb}_i := \text{rgb}_i \setminus \{\text{Req}(v, v', o, p-1)\} \\ \text{rgb}_{i+1} := \text{rgb}_{i+1} \cup \{\text{Req}(v, v', o, p)\} \end{array} \right\} \quad (T6)$$

$$\text{Res}_i(v, v', \text{acc}) :$$

$$\left\{ \begin{array}{l} \text{Req}(v, v', o, p) \in \text{rgb}_i \wedge \\ \neg \text{voted}_i \wedge o+1 > n/2 \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} \text{rgb}_i := \text{rgb}_i \setminus \{\text{Req}(v, v', o, p)\} \\ \text{rsb}_k := \text{rsb}_k \cup \{\text{Res}(v, v', \text{acc})\} \quad \text{all } k \end{array} \right\} \quad (T7)$$

$$\text{Res}_i(v, v', \text{rej}) :$$

$$\left\{ \begin{array}{l} \text{Req}(v, v', o, p) \in \text{rgb}_i \wedge \\ v < x_i \vee \left(\begin{array}{l} \text{voted}_i \wedge \\ (v, v') <_{\text{prio}} \text{voteval}_i \wedge \\ (p+1 \geq n/2) \end{array} \right) \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} \text{rgb}_i := \text{rgb}_i \setminus \{\text{Req}(v, v', o, p)\} \\ \text{rsb}_k := \text{rsb}_k \cup \{\text{Res}(v, v', \text{rej})\} \\ \text{all } k \end{array} \right\} \quad (T8)$$

$$\tau : \text{Res}(v, v', r) \in \text{rsb}_i \longrightarrow \left\{ \begin{array}{l} \text{rsb}_i := \text{rsb}_i \setminus \{\text{Res}(v, v', r)\} \\ \text{if } r = \text{acc} \wedge v' > x_i \text{ then } x_i := v' \\ \text{if } (v, v') = \text{voteval}_i \text{ then } \text{voted}_i := \text{false} \\ \text{if } i = \text{origin}(v') \text{ then } \text{nfb}_i := \text{nfb}_i \cup \{\text{Notify}(v, v', r)\} \end{array} \right\} \quad (T9)$$

Fig. 5.3.4. Transitions of $\Pi(\text{DBM}_1, \dots, \text{DBM}_n)$.

To establish the above implication, we must find a simulation R between $\Pi(\mathcal{A}(DBM_0), \dots, \mathcal{A}(DBM_{n-1}))$ and $\mathcal{A}(DB)$, which satisfies the requirements in theorem 4.4.6. In the simulation R , the variables x_i , q_i , and $stamp_i$ have the same values in $\Pi(\mathcal{A}(DBM_0), \dots, \mathcal{A}(DBM_{n-1}))$ as in $\mathcal{A}(DB)$. This is easily verified by inspecting the initial predicates and transitions of the involved trace generators. We shall therefore not bother to distinguish their occurrences in $\Pi(\mathcal{A}(DBM_0), \dots, \mathcal{A}(DBM_{n-1}))$ from those in $\mathcal{A}(DB)$. The relation R also contains the following assertions.

$$(\exists i, o, p)[Req(v, v', o, p) \in rqbi] \equiv Req(v, v') \in rqb \quad (R1)$$

$$\left\{ \begin{array}{l} Notify(v, v', r) \in nfb_{origin(v')} \\ \vee Res(v, v', r) \in rsb_{origin(v')} \end{array} \right\} \equiv Notify(v, v', r) \in nfb \quad (R2)$$

$$Notify(v, v', r) \in nfb_i \supset i = origin(v') \quad (R3)$$

$$(\forall i, v') \left[\begin{array}{l} \sum_{j, v, o, p} \# Req(v, v', o, p) \text{ in } rqbi + \\ \sum_{v, r} \# Res(v, v', r) \text{ in } rsbi + \\ \sum_{v, r} \# Notify(v, v', r) \text{ in } nfb_i \end{array} \right] \leq 1 \quad (R4)$$

$$[x_i = v' \vee Res(v, v', acc) \in rsbi] \supset v' \in xset \quad (R5)$$

$$[v' \in xset \wedge x_i < v'] \supset (\exists v)[Res(v, v', acc) \in rsbi] \quad (R6)$$

$$\left[\begin{array}{l} Req(v, v') \in rqb \vee \\ Res(v, v', r) \in rsb \vee \\ Notify(v, v', r) \in nfb \end{array} \right] \supset \left[\begin{array}{l} v \leq latest(xset) \wedge \\ v' \leq stamp_{origin(v')} \wedge \\ v < v' \end{array} \right] \quad (R7)$$

$$Req(v, v', o, p) \in rqbi \supset o = \left(\sum_{j=0}^{n-1} [voted_j \wedge voteval_j = (v, v')] \right) \quad (R8)$$

$$\sum_{i=0}^{n-1} \left(\begin{array}{l} \left[\begin{array}{l} x_i = latest(xset) \wedge \\ voted_i \supset (\exists v')[voteval_i = (x_i, v')] \end{array} \right] \vee \\ (\exists u) \left[\begin{array}{l} voted_i \wedge voteval_i = (u, latest(xset)) \wedge \\ Res(u, latest(xset), acc) \in rsbi \end{array} \right] \end{array} \right) > n/2 \quad (R9)$$

Assertions (R1) - (R4) describe the relationship between the bags $reqb$ and nfb of $\mathcal{A}(DB)$, and the bags $reqb_i$, rsb_i , and nfb_i of each $\mathcal{A}(DBM_i)$. Assertion (R1) states that the bag of request messages in $reqb$ contains exactly the request messages of all $reqb_i$. Assertion (R2) states that the pending notifications in nfb correspond either to resolution messages in rsb_i or to pending notifications in nfb_i , where i is the site where the request originated. Assertion (R3) states that notification messages for update requests must only be at the manager where the request originated. Assertion (R4) states that a request for an update is either in at most one of the sets rsb_i or nfb_i for each i (if it has been accepted or rejected), or in at most one of the sets $reqb_j$ (if it has not yet been accepted or rejected).

Assertions (R5) - (R6) describe the relation between the local copy x_i of the datum and the other variables. Assertion (R5) states that each local value x_i or new value in a pending resolution message in rsb_i is among the values in $xset$. Assertion (R6) states that if the local value x_i is less recent than some value v' in $xset$, then the value v' is a new value in a pending resolution message in rsb_i .

Assertion (R7) concerns the timestamps of the request messages in the database. It states that for a request message in the database with a proposed update from v to v' , it holds that $latest(xset)$ is at least as recent as v (i.e., that the proposed update updates a value which exists or may have existed in the database), that $v' \leq stamp_{origin}(v')$, i.e., that $stamp_{origin}(v')$ correctly reflects the latest proposed update, and that the new value is more recent than the old value.

Assertion (R8) states that the number of OK votes on a request message is the number of managers that have voted for this request. To express this property we arithmetize predicates, taking *true* equal to 1, and *false* equal to 0.

The most important assertion is (R9). it states that more than $n/2$ of the managers either

- (a) have the current copy of the datum, i.e., the latest value in $xset$, and have either no pending vote, or a vote given after the last update, or
- (b) have voted for the last update, i.e., the update to $latest(xset)$, but have not processed the corresponding resolution message.

Also here, we also use arithmetization of predicates.

We must verify that R is indeed a simulation. We shall verify that (T1) is simulated by (s1), (T2) by (s2), (T3) by (s3), (T4) by (s4), (T5) and (T6) by stuttering transitions, (T7) by (s5), and that (T8) is simulated by (s6). The transition (T9) is simulated by (s7) if $r = acc \wedge v' > x_i$, otherwise by a stuttering transition.

Initially, R holds trivially. It is easy to verify that R is preserved by the transitions $(T1)$, $(T2)$, and $(T4)$. The transition $(T3)$ adds a request message of form $Req(v, v', o, p)$ to the database, in which case we must verify that $(R6)$ and $(R7)$ are preserved. The conjunct $(R6)$ is preserved since if any other request message $Req(u, u', o', p')$ is previously in the database, then $u' \neq v'$ follows from the fact that if $u' = v'$ then $origin(u') = origin(v')$, which by $(R7)$ and the precondition of $(T3)$ gives $u' \leq stamp_{origin(v')} < v'$, a contradiction. To verify that the conjunct $(R7)$ is preserved, we check the conjuncts in its consequent for the new request message. The first conjunct follows from $(R5)$ and the enabling condition of $(T3)$ which imply $v \leq x_i \leq latest(xset)$. The second conjunct follows by the way $(T3)$ updates $stamp_i$. The third conjunct follows from the enabling condition of $(T3)$.

For the two voting transitions $(T5)$ and $(T6)$, we must check that $(R8)$ is preserved. This follows easily.

The heart of the proof is the transition $(T7)$: verifying that updates are done correctly. We must prove that the enabling condition of $(T7)$ implies the enabling condition

$$\{Req(v, v') \in rqb \wedge v = latest(xset)\}$$

of $(s5)$. The first conjunct follows from $(R1)$. For the second conjunct, note that $(R8)$ and the precondition of $(T7)$ imply that more than $n/2$ managers vote for $Req(v, v')$. Also note that $(R9)$ states a property which is satisfied by more than half of the managers. It follows that there must be one DBM_j which both has voted for $Req(v, v')$ (i.e., $voted_j$ and $voteval_j = (v, v')$) and either

- (a) $x_j = latest(xset)$ and $voteval_j = (x_j, v')$ (since $voted_j$ is true according to $(R8)$). It follows that $v = latest(xset)$.
- (b) there is a value u such that $voteval_j = (u, latest(xset))$ and rsb_j contains the resolution message $Res(u, latest(xset), acc)$. By conjunct $(R4)$ this contradicts the fact that $Req(v, v', o, p) \in rqb_i$, since a request which is in some rqb_i cannot also be in some rsb_j . This takes us back to case (a).

To verify that R holds after $(T7)$, note that the added resolution message $Res(v, v', acc)$ accounts both for the extension of $xset$ and of nfb .

Transition $(T8)$ trivially preserves R , since one pending request is discarded from rqb_i and from rqb . For transition $(T9)$, note that in the case when $r = acc$ and $v' > x_i$ we can take v' in $(s7)$ equal to v' in $(T9)$ to satisfy the first conjunct in the enabling condition of $(s7)$. The second conjunct in the enabling condition of $(s7)$ follows from $(R5)$, which states that v' must be a member of $xset$ if the condition of $(T9)$ is satisfied.

We next turn to the verification of liveness requirements of DB . It is trivial

to verify that (L1) follows from the requirements (L_i1), and that (L2) follows from the requirements (L_i2).

The requirement (L3), follows from the liveness requirements (L_i3), since the conjunct (R6), the assertion $x_i \leq v \wedge v \in xset$, implies that $Res(v', v, acc) \in rsb_i$. Now we use liveness requirement (L_i3) to conclude that a transition of type (T9) will occur in which x_i is updated.

Requirement (L4) is verified using the chain rule. Consider a request message $R(v, v', o, p)$ in some rqb_i . Since $o + p$ votes have been given to this message, there are $n - (o + p)$ managers that have not yet considered it. In the proof of (L4), we shall perform induction on the sum of possible future considerations over all request messages. This sum is the parameter w of the chain rule. Thus, we define $\chi(w)$ as

$$R \wedge (\exists v, v') \left[\begin{array}{l} Req(v, v') \in rqb \wedge \\ v \geq latest(xset) \end{array} \right] \wedge w = \sum_{Req(v, v', o, p) \in rqb_j} (n - (o + p))$$

Note that w is always a nonnegative integer. We use the ordinary less-than relation between integers as the well-founded order relation $<$. We shall verify that

$$(\chi(w)) \xrightarrow[R]{\mathcal{T}} \diamond \left((latest(xset^{new}) > latest(xset^{old})) \vee (\exists i) Update_i ; \right. \\ \left. (\exists w' < w) [\chi(w')] \right)$$

To prove this formula, we use the guarantee rule with the guarantee requirement (L_i4). We take parameters v, v' of (L_i4) such that $Req(v, v')$ is the request message in rqp which is still valid (i.e., $latest(xseq) = v$), but has lowest priority among the valid requests. (L_i4) states that this request message will eventually be processed, unless no request message with lower priority arrives, but in that case $(\exists i) Update_i$ is satisfied. More precisely, the premises of the guarantee rule are verified as follows:

- (1) Unless an $Update_i$ event occurs, or $latest(xset)$ is updated, the number w of still possible votes can not increase. Also, there will still be a valid request message in rqb , since no PASS votes can be put on the message with highest priority.
- (2) When the valid request message with lowest priority is considered, then either some vote is put onto it, decreasing w , or some decision is taken. If the decision is *reject*, then w decreases. There will still be a valid request message, since the request message with highest priority cannot be rejected. If the decision is *accept*, then $xset$ is updated.
- (3) Follows from the discussion above.

5.3.5 Discussion

We have verified safety and liveness properties of an algorithm, due to Thomas ([Th 79]) for ensuring consistency in a distributed database. Our description of the algorithm follows rather closely that by Thomas. We made the simplification to consider only one data element in the database. Thomas also extends the algorithm to consider the possibility of communication failures, in which case retransmissions must be used. In this case, the algorithm must be revised. We have not considered this extension.

Thomas presents a rather detailed informal proof that the algorithm works as desired. The main part of the proof consists of verifying that updates occur consistently. By induction, he proves that the accepted updates can be put in a linear sequence such that each accepted update operates on the values of the previous update. This argument is fairly long. We feel that our statement and proof of this aspect, which is mainly described by transition (s5) in Fig. 5.3.3, is clear and succinct.

Thomas also proves that each update request will eventually be either accepted or rejected by the database. This depends crucially on the fact that voting on a request message cannot be deferred indefinitely. For this, one must ensure that only a bounded number of request messages can be generated with a timestamp which is less than a certain value. This follows for instance by assuming that timestamps are integer-valued. We could have established this liveness property by a method similar to the one used when establishing (L4). Thomas does not establish liveness property (L4), as we do.

Safety properties of this algorithm have also been specified and verified by Ossefort ([Os 82]), using a method by Misra and Chandy ([MC 81]). Ossefort uses predicates and functions over traces. It is difficult to compare directly the two approaches.

6 Conclusion

In this chapter, we briefly summarize the contributions of the thesis, indicate some connections with related work, and point out some directions for future research.

Summary

We have presented a method for modeling, specification, and verification of message-passing distributed systems. The method applies to systems that can always accept input from other systems. The method is compositional and handles both safety and liveness properties.

The semantical foundations for the method were provided by a model, which describes a system by the set of its quiescent traces. Based on this model, we presented a method for specifying systems by properties of their quiescent traces. These properties are expressed by transition systems with liveness requirements. We presented proof rules for verifying the specification of a system from specifications of its components. Safety properties are verified by establishing a simulation between transition systems. Liveness properties are verified using induction over well-founded sets.

We have also investigated the relationship between our model and other models of distributed systems. For nondeterministic Kahn-networks, we have shown that our model is the minimal compositional extension of the history model, in the sense of containing least information.

We illustrated the method by applying it to the verification to three examples from the literature: The example of Brock and Ackermann, the sliding window protocol found in HDLC, and Thomas' algorithm for ensuring consistency in a distributed database.

Related models

Our investigations of the relationship between models of I/O-systems is partly motivated by the abundance of models of distributed systems. Olderog and Hoare ([OH 84]) have investigated models of communicating sequential processes and their relation to each other.

The results in this thesis only concern a few models of I/O-systems. Some questions for further study are:

- Is there, in addition to the safety model, another compositional model which is strictly more informative than the event model, but more abstract than the liveness model?
- How is the existence of compositional models affected by the class of systems considered? We could for instance investigate compositional models of deterministic I/O-systems, deterministic and nondeterministic Kahn-networks, etc.

Related specification and verification methods

Approaches to specification and verification that are related to ours are presented by Lamport ([La 83]) and Stark ([St 84], [St 86]). Both use transition systems with liveness requirements to specify systems.

Stark uses transition systems to specify the allowed sequences of communication events of a system, as in our approach. Lamport uses another criterion for what it means for a system to satisfy a specification. In his approach, the transition system expresses constraints on the transitions of a specified system, not on sequences of communication events. Lamport and Stark specify liveness properties in linear temporal logic. Stark formulates liveness properties in a certain style, as rely-guarantee conditions ([St 85]).

To verify safety properties of a system, Stark establishes a simulation between transition systems, in a way similar to ours. Lamport establishes a mapping between the states of transition systems. When verifying liveness properties, Lamport and Stark show that the liveness properties of one transition system follow from the liveness properties of the implementation.

In our method, liveness requirements have a more restricted form than in the approaches of Lamport and Stark. Our method for verifying liveness requirements of a specification uses induction over a well-founded set, similar to proof methods commonly used for proving termination of programs. The intention is that the method should provide guidelines for finding and structuring the proof. It may also preclude logical errors in the verification by disallowing certain types of reasoning that are more likely to lead to mistakes.

In our earlier work ([Jo 85]), we specified properties of quiescent traces using functions and predicates over sequences. In the work by Hailpern and Owicki ([HO 83]) and by Nguyen et al ([NDGO 86]), properties of sequences are specified by temporal logic operators and predicates over finite sequences. In these approaches the proof system is related to the proof system of section 4.2.

Specification and verification by transition systems

Our specification method uses transition systems. When applying the method to examples, we have found the notion of state helpful when formulating a specification of a complex pattern of interaction, as for instance in the database example. On the other hand, certain systems can be more concisely specified in other formalisms. In example 4.2.5, we verified that the composition of two unbounded FIFO buffers satisfy the specification of a single FIFO buffer, using projection functions and equality between sequences. This verification is simpler than the corresponding treatment using transition systems in example 4.4.8. This observation partly motivated the techniques in section 4.6 for specification and verification of systems that communicate via FIFO buffers.

When employing a specification formalism with states and transitions, specifications look rather similar to programs. This can be an advantage when actually implementing a system, since the specification can provide guidelines for the design of the implementation. On the other hand, the specification may put too many restrictions on the system, and preclude a better implementation which departs from the structure of the transition system in the specification. Care is needed to formulate a specification which is as general as possible without introducing unnecessary constraints. Schwartz and Melliar-Smith ([ScM 82]) discuss the use of states in specifications of communication protocols.

Incompleteness of proof method

Our proof rules for verification of properties given by transition systems reduce a proof about sequences of transitions to a proof which only considers individual transitions. As noted in section 4.5, the method is not complete.

It seems that for transition systems, it is more difficult to establish an inclusion between sequences of events generated by sequences of transitions than to find a simulation between the states of the transition systems. In another framework, Kanellakis and Smolka ([KS 83]), have shown that the problem of finding a bisimulation between two finite-state automata is solvable in polynomial time (a bisimulation is a simulation whose inverse is also a simulation). The corresponding problem for entire computations, that of deciding whether two finite-state automata accept the same language, has been shown to be PSPACE-complete by Stockmeyer and Meyer ([StM 73]).

The restrictions on the result about semantical completeness in theorem 4.5.3 suggests that it is advisable to avoid nondeterminism in the transition systems whenever possible. However, as illustrated by the database example, nondeterminism caused by silent transitions occur naturally in many specifications.

Specifying communication media

We have applied our method to the specification and verification of systems that communicate by passing messages over an intermediate medium. In our method, we specify a system together with the medium used for transmitting messages to the system. For example, in the Brock-Ackermann example, a specification of a component also specifies the FIFO channel used to receive input messages.

An alternative approach is to omit the network from the specification of components, and to specify the medium as a separate component. In general, we cannot formulate this approach using our method, since a component by itself may sometimes be unable to receive messages from the medium. These two approaches should be compared by applying them to examples.

Semi-automatic tools

Semi-automatic tools should be developed that can assist in a verification performed by a human. To develop such tools, the verification method must be developed into a form suitable for implementation.

References

- [AFR 80] Apt, K., Francez, N., and de Roever, W. "A proof system for communicating sequential processes." *ACM Trans. on Programming Languages and Systems* 2, 3 (1980), pp. 359-385.
- [BM 82] Back, R.J.R., and Mannila, H. "A refinement of Kahn's semantics to handle non-determinism and communication." *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1982, pp. 111-120.
- [BM 85] Back, R.J.R., and Mannila, H. "On the suitability of trace semantics for modular proofs of communicating processes." *Theoretical Computer Science* 39, 1 (1985), pp. 47-68.
- [BC 87] Boudol, G., and Castellani, I. "On the semantics of concurrency, partial orders and transition systems." in Ehrig, Kowalski, Levi, Montanari eds. *TAPSOFT '87, Lecture Notes in Computer Science* 249, Springer Verlag, 1987, pp. 123-137.
- [Bo 82] Boussinot, F. "Proposition de sémantique dénotationnelle pour des processus avec opérateur de mélange équitable." *Theoretical Computer Science* 18, 2 (1982), pp. 173-206.
- [BJ 82] Brand, D., and Joyner, W.H. "Verification of HDLC." *IEEE Trans. on Communications*, COM-30, 5 (1982), pp. 1136-1142.
- [Br 79] Brauer, W. ed. *Net Theory and Applications, Lecture Notes in Computer Science* 84, Springer Verlag, 1979.
- [BA 81] Brock, J.D., and Ackerman, W.B. "Scenarios: a model of non-determinate computation." In Diaz, Ramos eds. *Formalization of Programming Concepts, Lecture Notes in Computer Science* 107, Springer Verlag, 1981, pp. 252-259.
- [Brookes 83] Brookes, S.D. A Model for Communicating Sequential Processes, CMU-CS-83-149, Carnegie-Mellon University, 1983.
- [BHR 84] Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. "A theory of communicating sequential processes." *ACM Journal* 31, 3 (1984), pp. 560-599.
- [Broy 83] Broy, M. "Fixed point theory for communication and concurrency." in Bjoerner ed. *Formal Description of Programming Concepts II*, North Holland, Amsterdam, 1983, pp. 125-146.
- [CH 81] Chen, Z.C., and Hoare, C.A.R. "Partial correctness of communicating processes and protocols." Technical monograph PRG-20, Programming Research Group, Oxford Computing Laboratory, 1981.
- [Di 76] Dijkstra, E.W. *A Discipline of Programming*, Prentice-Hall, 1976.
- [Fl 67] Floyd, R. "Assigning meanings to programs." in Schwartz, ed. *Proc. ACM Symp. on Applied Mathematics* 19, 1967, pp. 19-32.

- [Fr 86] Francez, N. *Fairness*, Springer Verlag, 1986.
- [FLP 84] Francez, N., Lehmann, D., and Pnueli, A. "A linear history semantics for languages for distributed programming." *Theoretical Computer Science* 32, 1 (July 1984), pp. 25-46.
- [GFMR 81] Grumberg, O., Francez, N., Makowsky, J.A., and de Roever, W.-P. "A proof rule for fair termination of guarded commands." *Proc. Int. Symp. on Algorithmic Languages*, Amsterdam, North-Holland, 1981, pp. 399-416.
- [HO 83] Hailpern, B.T., and Owicki, S. "Modular verification of computer communication protocols." *IEEE Trans. on Communications COM-31*, 1 (1983), pp. 56-68.
- [He 85] Hennessy, M. "Acceptance trees." *ACM Journal* 32, 4 (1985), pp. 896-928.
- [HMSTK 84] Higashino, T., Mori, M., Sugiyama, Y., Taniguchi, K., and Kasami, T. "An algebraic specification of HDLC procedures and its verification." *IEEE Trans. on Software Engineering*, 10, 6 (1984), pp. 825-836.
- [Ho 69] Hoare, C.A.R. "An axiomatic basis for computer programming." *ACM Communications* 12, 10 (1969), pp. 576-583.
- [Ho 78] Hoare, C.A.R. "Communicating sequential processes." *ACM Communications* 21, 8 (1978), pp. 666-676.
- [Ho 81] Hoare, C.A.R. "A calculus for total correctness for communicating processes." *Science of Computer Programming* 1, 1 (1981), pp. 49-72.
- [ISO 79] International Standards Organization, *Data Communications - HDLC Procedures - Elements of Procedures*, Ref. No. ISO 4335, International Standards Organization, Geneva, Switzerland, 1979.
- [Jo 85] Jonsson, B. "A model and proof system for asynchronous networks." *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 49-58.
- [Ka 74] Kahn, G. "The semantics of a simple language for parallel programming." *Proc. IFIP 74*, North-Holland, Amsterdam, 1974, pp. 471-475.
- [KS 83] Kanellakis, P.C., and Smolka, S.A. "CCS expressions, finite state processes, and three problems of equivalence." *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, 1983, pp. 228-240.
- [Ke 78] Keller, R.M. "Denotational models for parallel programs with indeterminate operators." in Neuhold ed. *Formal Descriptions of Programming Concepts*, North-Holland, Amsterdam, 1978, pp. 337-366.
- [KP 84] Keller, R.M., and Panangaden, P. "Semantics of networks containing indeterminate operators." In Brookes, Roscoe, Winskel, eds. *Proc. Seminar on Concurrency, 1984, Lecture Notes in Computer Science 197*, Springer Verlag, 1985, pp. 479-496.
- [Ko 86] Kok, J.N. "Denotational semantics of nets with nondeterminism." in *European Symposium on Programming, Saarbrücken, Lecture Notes in Computer Science 206*, Springer Verlag, 1986, pp. 237-249.
- [Ko 78] Kosinski, P.R. "A straight-forward denotational semantics for nondeterminate data flow programs." *Proc. 5th ACM Symp. on Principles of Programming Languages*, 1978, pp. 471-475.
- [La 83] Lamport, L. "Specifying concurrent program modules." *ACM Trans. on Programming Languages and Systems* 5, 2 (1983), pp. 190-222.
- [LPS 81] Lehman, D., Pnueli, A., and Stav, J. "Impartiality, justice and fairness: the ethics of concurrent termination." In Even, Kariv eds. *Proc. Int. Coll. on Automata, Languages and Programming, Lecture Notes in Computer Science 115*, Springer Verlag, 1981, pp. 264-277.
- [Lø 85] Løvgreen, H.H. "On concurrency formalization." ID-TR 1985-3, Inst. f. Data-teknik, Danmarks Tekniske Højskole, Denmark, 1985.

- [LF 81] Lynch, N.A., and Fischer, M.J. "On describing the behavior and implementation of distributed systems." *Theoretical Computer Science* 13, 1 (1981), pp. 17-43.
- [MP 81] Manna, Z.M., and Pnueli, A. "The temporal framework for concurrent programs." in Boyer, Moore, eds. *The Correctness Problem in Computer Science*, Academic Press, 1981, pp. 215-274.
- [MP 84] Manna, Z., and Pnueli, A. "Adequate proof principles for invariance and liveness properties of concurrent programs." *Science of Computer Programming* 4, 4 (1984) pp. 257-289.
- [Ma84] Mazurkiewicz, A. "Traces, histories, graphs: instances of a process monoid." in Chytil, Koubek eds., *Mathematical Foundations of Computer Science 1984, Lecture Notes in Computer Science 176*. Springer Verlag, 1984, pp. 115-133.
- [Mil 80] Milner, R. "A calculus of communicating systems." *Lecture Notes in Computer Science* 92, Springer Verlag, 1980.
- [Mis 84] Misra, J. "Reasoning about networks of communicating processes." INRIA Advanced Nato Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, La Colle sur Loupe, France, 1984.
- [MC 81] Misra, J., and Chandy, K.M. "Proofs of networks of processes." *IEEE Trans. on Software Engineering* SE-7, 4 (1981), pp. 417-426.
- [MCS 82] Misra, J., Chandy, K.M., and Smith, T. "Proving safety and liveness of communicating processes with examples." *Proc. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1982, pp. 201-208.
- [NDGO 86] Nguyen, V., Demers, A., Gries, D., and Owicki, S. "A model and temporal proof system for networks of processes." *Distributed Computing* 1, 1 (1986), pp. 7-25.
- [OH 84] Olderog, E.R., and Hoare, C.A.R. "Specification-oriented semantics for communicating processes." Report PRG-37, Programming Research Group, Oxford University, 1984.
- [Os 82] Ossefort, M. A Unified Approach to Formal Verification of Network Safety Properties, Ph.D Dissertation, Dept of Computer Sciences, Univ. of Texas, Austin, 1982.
- [OG 76] Owicki, S., Gries, D. "Verifying properties of parallel programs: an axiomatic approach." *ACM Communications* 19, 5 (1976), pp. 279-284.
- [OL 82] Owicki, S., Lamport, L. "Proving liveness properties of concurrent programs." *ACM Trans. on Programming Languages and Systems* 4, 3 (1982), pp. 455-495.
- [Pac 82] Pachl, J.K. Reachability Problems for Communicating Finite State Machines, CS-82-12, Faculty of Math., Univ. of Waterloo, Ontario, 1982.
- [Pa 83] Park, D. "The 'fairness' problem and nondeterministic computing networks." In de Bakker, Leuwen eds. *Foundations of Computer Science IV, Part 2*, Mathematical Centre Tracts 159, Amsterdam 1983, pp. 133-161.
- [Pe 81] Peterson, J.L. *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.
- [Pl 81] Plotkin, G.D. "A structural approach to operational semantics." DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.
- [Pr 82] Pratt, V.R. "On the composition of processes." *Proc. 9th ACM Symp. on Principles of Programming Languages*, 1982, pp. 213-223.
- [Pr 84] Pratt, V.R. "The pomset model of parallel processes: unifying the temporal and the spatial." In Brookes, Roscoe, Winskel, eds. *Proc. Seminar on Concurrency, 1984, Lecture Notes in Computer Science 197*, Springer Verlag, 1985, pp. 180-196.
- [Pr 85] Pratt, V.R. "Some constructions for order-theoretic models of concurrency." In Parikh ed. *Proc. Logics of Programs, 1985, Lecture Notes in Computer Science 193*, Springer Verlag, 1985.

References

- [ScM 82] Schwartz, R.L., and Melliar-Smith, M. "From state machines to temporal logic: specification methods for protocol standards." *IEEE Trans. on Communications COM-30*, 12 (1982), pp. 2486-2496.
- [SL 83] Shankar, K.S., and Lam, S.S. "An HDLC protocol specification and its verification using image protocols." *ACM Trans. on Computer Systems* 1, 4 (1983), pp. 331-368.
- [SN 85] Staples, J., and Nguyen, V.L. "A fixpoint semantics for nondeterministic data flow." *ACM Journal* 32, 2 (April 1985), pp. 411-444.
- [St 84] Stark, E.W. Foundations of a Theory of Specification for Distributed Systems, Ph.D. Thesis, MIT/LCS/TR-342, Massachusetts Inst. of Technology, 1984.
- [St 85] Stark, E.W. "A proof technique for rely/guarantee properties." in *Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 206*, Springer Verlag, 1985, pp. 369-391.
- [St 86] Stark, E.W. "Proving entailment between conceptual state specifications." in *European Symposium on Programming, Saarbrücken, Lecture Notes in Computer Science 206*, Springer Verlag, 1986, pp. 197-209.
- [StM 73] Stockmeyer, L.J., and Meyer, A.R. "Word problems requiring exponential time." *Proc. 5th ACM Symp. on Theory of Computing*, 1973, pp. 1-9.
- [Th 79] Thomas, R.H. "A majority consensus approach to concurrency control for multiple copy databases." *ACM Trans. on Database Systems* 4, 2 (1979), pp. 180-209.
- [WGS 87] Widom, J., Gries, D., and Schneider, F.B. "Completeness and incompleteness of trace-based network proof systems." *14th ACM Symp. on Principles of Programming Languages*, 1987, pp. 27-38.
- [Wi 80] Winskel, G. Events in Computation, Ph.D. Thesis, Dept. of Computer Science, Univ. of Edinburgh, 1980.

ISSN 0283-0574

Direkt Offset, Nyström & Co AB, Uppsala 1987